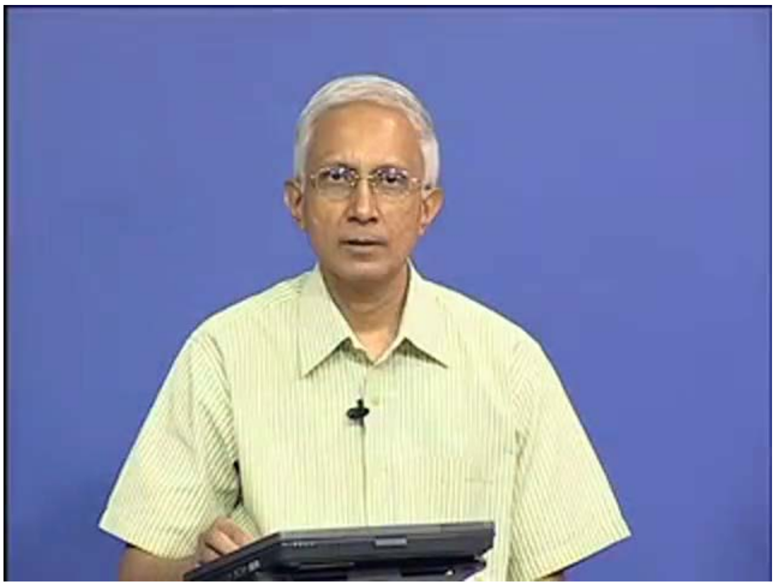
High Performance Computing

Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 06
Lecture No. # 30

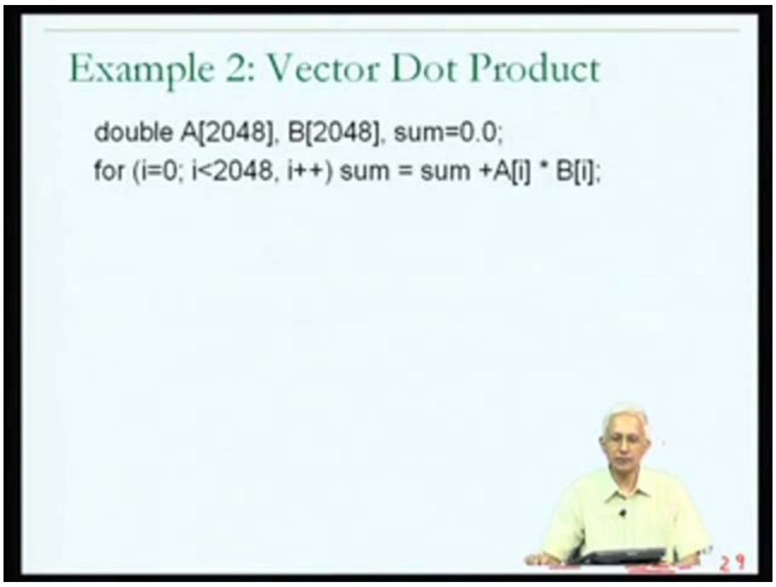
(Refer Slide time: 00:14)



In the previous lecture, we completed understand our discussion of the different cache organizations and we started looking at some small programming examples to get a better appreciation of how small changes in a program or in a loop or important loop of a program, might affect the cache behavior the cache performance that the program would benefit from when executed on a computer with cache memories. And we were in the process of looking at a second example which I called vector dot product. Now, all of the examples that we are looking at relatively small in terms of the number of lines of c code, but they may be important in that they could be fairly frequently occurring operations in larger programs and therefore, a proper understanding of such small, but possibly important loops is beneficial to us.

So, the second example which we are we are looking at it is what I called vector dot product and in the slide we had looked at the c version of the vector dot product.

(Refer Slide time: 01:19)



Example 2: Vector Dot Product

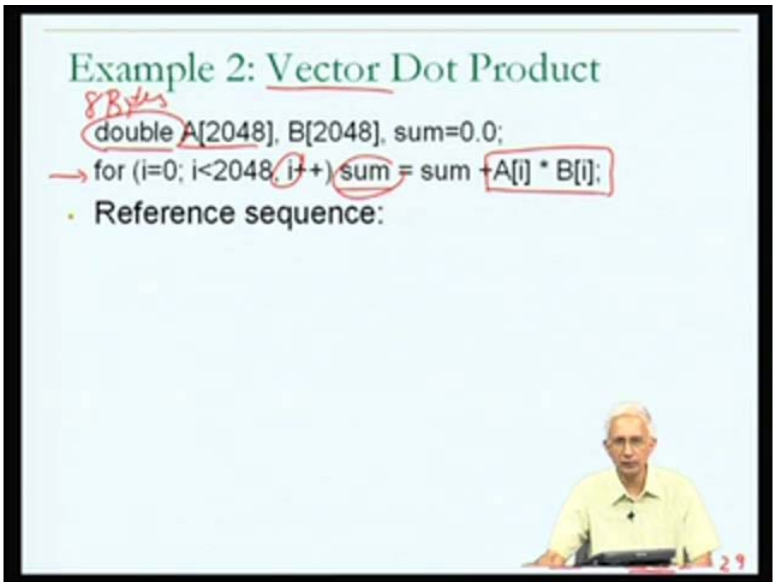
```
double A[2048], B[2048], sum=0.0;
for (i=0; i<2048, i++) sum = sum +A[i] * B[i];
```

So, as the name suggested it is a operation on vectors, which are one dimensional arrays of data. In this particular case there are two vectors which are multiplied term for term and the some of the products is the result of the dot product. So, we see the for loop in c where the dot product of the vector a and the vector b is computed by element by element multiplying a of i by b of i and accumulating the sum in a double procession variable, note once again that the both the vectors a and b are of size 2048, but more important they are each element of the vector is of type double procession or double float which means that the size of each element is eight bytes.

Now, the way that we were trying to understand this kind of an example, was first of all by ignoring all the instructions though we know that instructions have to be fetched from memory we assume, that the instructions will benefit from a separate instruction cache and that we can concentrate our analysis on the data behavior. Further we will ignore certain pieces of data such as the loop index or a variable like sum, which could quite easily be stored in registers by the compiler. And we are therefore, left only with a references to the array elements a and b which since there are 2048 of each it is unreasonable to assume that all of them can be accessed out of registers.

So, now the next step was to understands specifically, what this program what this loop would amount to in terms of load and store instructions in other words from the perspective of the cache for the memory system what does this program mean. And we analyze it in terms of what I have refer to as a reference sequence.

(Refer Slide time: 03:02)



Example 2: Vector Dot Product

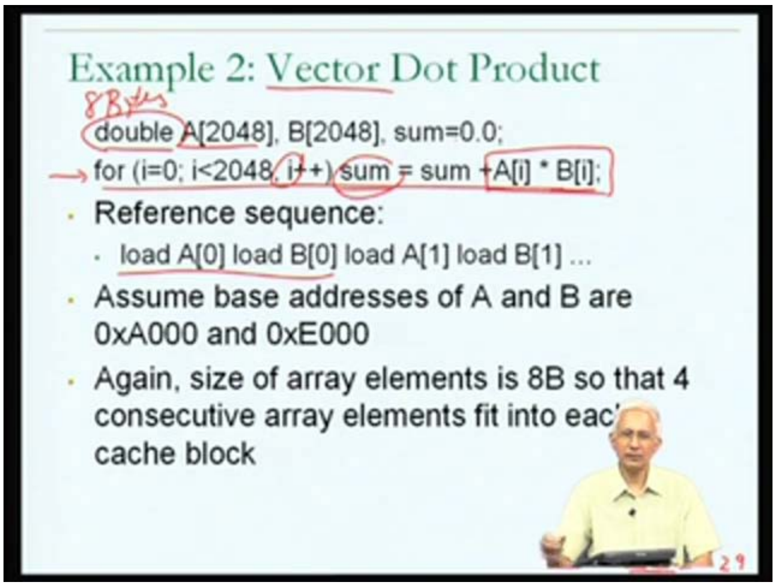
8 Bytes

```
double A[2048], B[2048], sum=0.0;
for (i=0; i<2048; i++) sum = sum +A[i] * B[i];
```

- Reference sequence:

The sequence of memory references that result when this loop executes and looking at a from the from the at the level of the arrays a and b, we realize that in the first iteration through the for loop. First of all a of zero will have to be loaded from memory into a register, then b of zero would have to be loaded from memory into a register subsequence, they would be multiplied, but we are not concerned about the multiply or the add or the other instructions we are concentrating only on loads and stores. Therefore, the reference sequence would simply have load a of zero load b of zero in the first iteration, in the second iteration of the for loop load a of one load b of one and so on.

So, all of the 2048, iterations would look a long glance of what we see here each iteration involving a load of an element of a and a load of corresponding element of the vector b. So, to proceed with our analysis we have to make some assumptions about what the addresses of the elements of a and b are.



Example 2: Vector Dot Product

8Bytes

```
double A[2048], B[2048], sum=0.0;
for (i=0; i<2048; i++) sum = sum + A[i] * B[i];
```

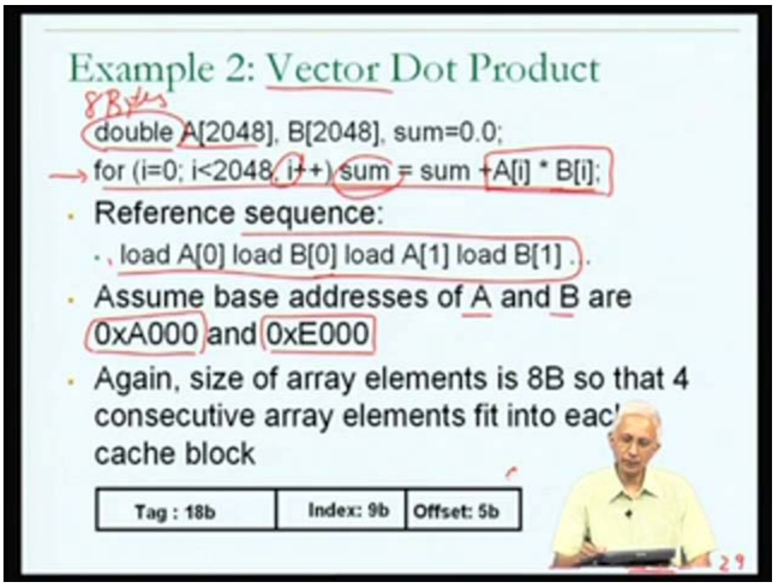
- Reference sequence:
  - load A[0] load B[0] load A[1] load B[1] ...
- Assume base addresses of A and B are 0xA000 and 0xE000
- Again, size of array elements is 8B so that 4 consecutive array elements fit into eac cache block

29

And the base assumption that we made is the compiler will assign an address to the first element of a into the first element of b and the subsequent elements of a and b would be in contiguous locations for example, a of one would be in the location next to a of zero. So, since each element of a is of size eight bytes if a of zero is at the address hex a is zero zero zero the day of one would be at the address hex a zero zero eight, which is eight bytes after or it is a contiguous in memory to the location of a of zero. And similarly, for all the other elements we similarly, assume, that the the compiler assigns b the vector b to some other address range and this information will be important to us in doing the cache in analysis.

Since, you will remember that to understand what is happening in the cache we have to keep track of the number of hits and misses when this program executes or when the sequence of memory references is sent to the cache the memory system. And therefore, we had to do the analysis based on how the cache hardware views an address and for the cache that we are considering in a in the examples, that we did last time we were assuming that there is a data cache of size sixteen kilobytes. The cache is direct mapped it uses write-back at the write-back update memory update policy and the size of each cache block is thirty two bytes. From this we could figure out exactly how the cache hardware views an address.

(Refer Slide time: 05:31).



Example 2: Vector Dot Product

8 Bytes

```
double A[2048], B[2048], sum=0.0;
for (i=0; i<2048; i++) sum = sum +A[i] * B[i];
```

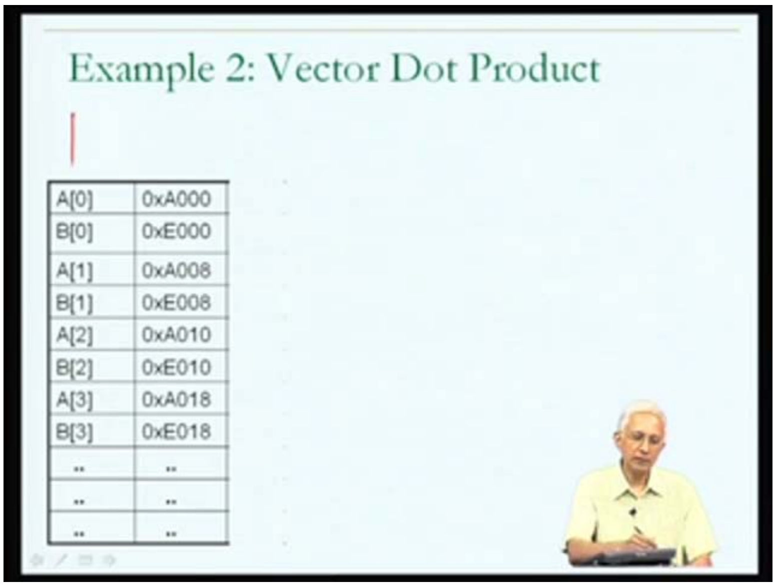
- Reference sequence:
  - load A[0] load B[0] load A[1] load B[1] ...
- Assume base addresses of A and B are 0xA000 and 0xE000
- Again, size of array elements is 8B so that 4 consecutive array elements fit into eac cache block

Tag : 18b	Index: 9b	Offset: 5b

29

So, if I look at the thirty two bits of an address such as the address the base address a of the base address of a vector b the least significant five bits are used as the offset to identify a particular byte within a block. The next significant nine bits are used index into the cache directory remember the we are assuming a direct map cache and the remaining bits are used by the cache hardware to distinguish between hits misses in terms of which main memory block is currently present in a given cache block. Therefore, this perspective for the particular cache that we are talking about the sixteen kilobyte cache with blocks of size thirty two bytes and using direct map placement is important for us to understand the behavior of the specific program that we are talking about.

(Refer Slide time: 06:21)



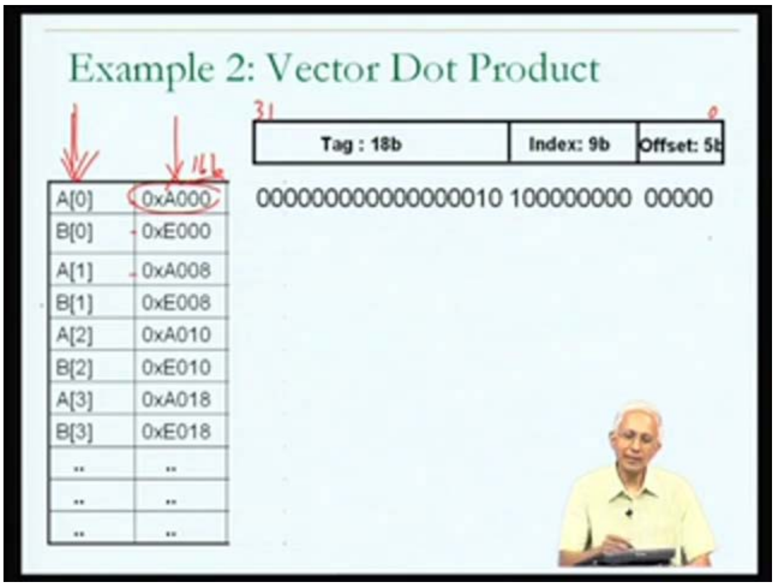
Example 2: Vector Dot Product

A[0]	0xA000
B[0]	0xE000
A[1]	0xA008
B[1]	0xE008
A[2]	0xA010
B[2]	0xE010
A[3]	0xA018
B[3]	0xE018
..	..
..	..
..	..

So, we went about doing the analysis by actually, deriving a table in which we had the different references the references are load a of zero load b of zero load a of one load b of one in the order as shown in the table. And we therefore, each of the references there is a an address for example, the address of a of zero is hex a zero zero zero. The address of b of zero we assuming is hex e zero zero zero the address of a of one as we just calculated is hex a 0, 0 eight and so on. Now, putting this together with our understanding of how the cache views addresses? The cache views addresses as least significant five bits remember these are thirty two bit addresses least significant five bits are the byte offset within a block and the next significant nine bits are the index into the cache directory.

So, in understanding what is going to happen within the cache when this program corresponding to these memory references executes, we have to view each of the addresses in the light of this cache perspective on addresses. So, consider a zero zero zero in hex if i take a zero zero and expand it into binary this is what I end up a thirty two bits sequence. Notice that a 0 0 0 each of the hex digits is actually, a four bit value and I have given you only four hex digits which means, that this is really a sixteen bit address. But I am expanding it into a thirty two bit address since that is what the cache hardware is assuming. And I do this by putting sixteen most significant zeros I do not want to change the value of the address therefore, hex a 0 0 0 0 is actually hex of 0 0 0 0 a 0 0 0 which is y I expanded in this way.

(Refer Slide time: 08:02)


Example 2: Vector Dot Product

Tag : 18b	Index: 9b	Offset: 5b
000000000000000010	100000000	00000

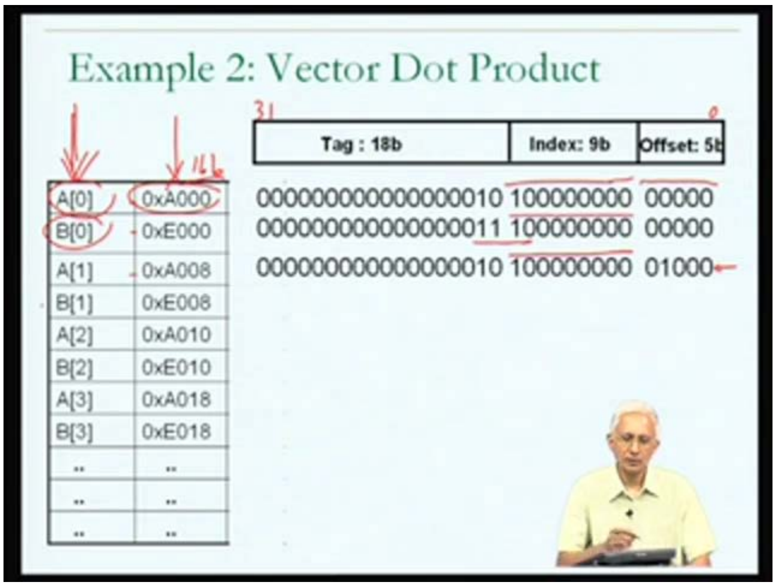
A[0]	0xA000
B[0]	0xE000
A[1]	0xA008
B[1]	0xE008
A[2]	0xA010
B[2]	0xE010
A[3]	0xA018
B[3]	0xE018
..	..
..	..
..	..

If I now, look at this thirty two bit version of hex a zero zero zero which is the base address of the vector a in terms of the least significant five bits the next significant nine bits. I find out that the index associated with a zero zero is the value 1 0 0 0 0 0 0 0 0 in binary, which you can compute to be equal to two fifty six. Similarly, I can look at the address the base address of b of zero which is hex e zero zero zero, which works out to the this bits sequence which I have expanded right now. So, the e is 111 0 in hex and the others of other bits are all zeros, but if I look at the index bits corresponding to this address I notice that once again it is exactly the same is the index bits associated with a of zero, which means that in my direct map cache.

The block containing vector element b of zero would occupy the same cache block as the block containing vector element a of zero this is a situation, the worst case situation that we talked about when we were worried about the problems with direct map caches and it has arises in this example. But if we continue to look through the sequence of references that we have what is the address of a of one the address of a of one as we saw is eight bytes eight more than the base address of a of zero. And that works out to the bit pattern which is third in sequence here it is eight added to the base address of the array and you will notice it too has an index of two fifty six in decimal and this we knew, because we know that a of one is a neighbor of a of zero and it would lightly be in the same cache block is a of zero and therefore, would have the same cache index.

We in fact from our calculation from last time given that the size of each cache block is thirty bytes and the size of each array element is eight bytes.

(Refer Slide Time: 10:00)



Example 2: Vector Dot Product

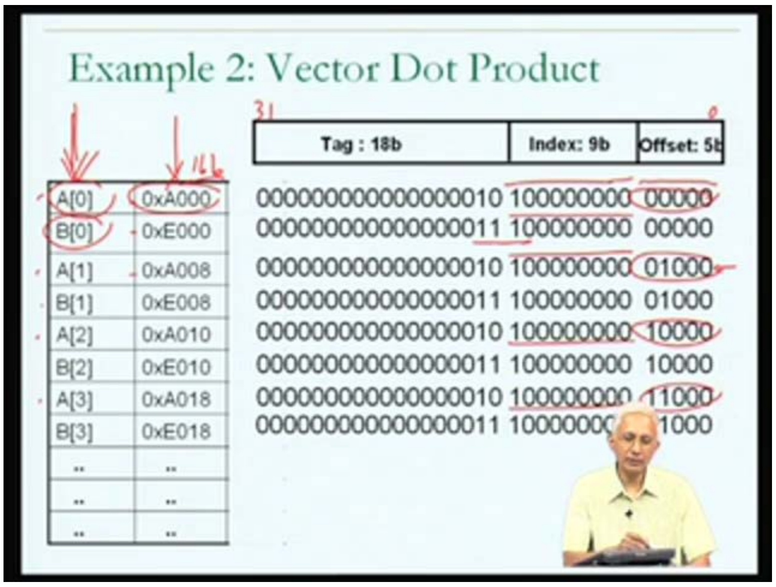
A[0]	0xA000
B[0]	0xE000
A[1]	0xA008
B[1]	0xE008
A[2]	0xA010
B[2]	0xE010
A[3]	0xA018
B[3]	0xE018
..	..
..	..
..	..

Tag : 18b	Index: 9b	Offset: 5b
000000000000000010	100000000	00000
000000000000000011	100000000	00000
000000000000000010	100000000	01000

We expect that four consecutive array elements whether it be for a of b four consecutive array elements would be in a single cache block. So, we expected that a of zero a of one a of two and a of three could all be within the same cache block in other words they would all have the same index. And as we proceed with expanding the addresses and looking at them in the light of the cache perspective on addresses, we do in fact see that is the case a of zero a of one a of two and a of three all have the same index. And differ only in their offset bits within the block they are talking about eight different corrections of eight bytes within the block of size thirty two bytes.

But the current problem that we have is very clearly after a of zero has been loaded a of zero, if we were assuming the cache starts of empty, then the attempt to load a of zero the first reference made by this vector dot product program would have resulted in a cache miss. And in handling the cache miss the cache hardware would have copied the block from main memory into the cache.

(Refer Slide time: 11:00)



Example 2: Vector Dot Product

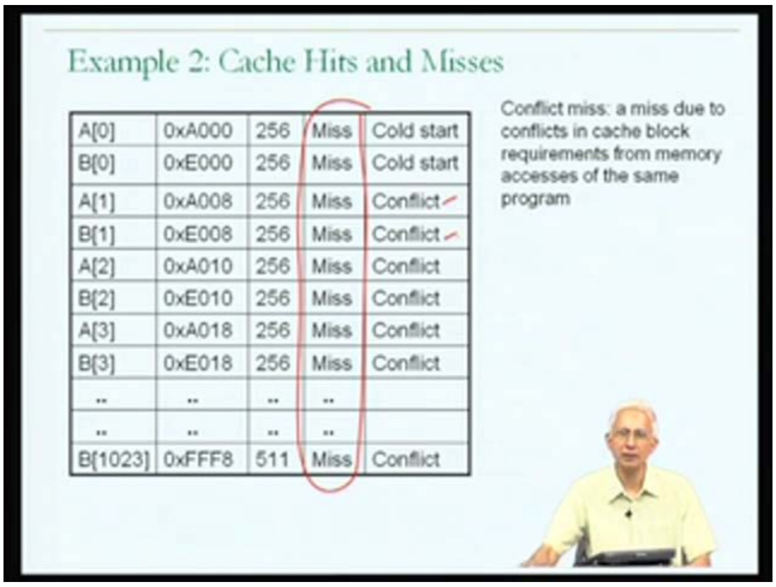
A[0]	0xA000
B[0]	0xE000
A[1]	0xA008
B[1]	0xE008
A[2]	0xA010
B[2]	0xE010
A[3]	0xA018
B[3]	0xE018
..	..
..	..
..	..

Tag : 18b	Index: 9b	Offset: 5b
000000000000000010	100000000	00000
000000000000000011	100000000	00000
000000000000000010	100000000	01000
000000000000000011	100000000	01000
000000000000000010	100000000	10000
000000000000000011	100000000	10000
000000000000000010	100000000	11000
000000000000000011	1000000	1000

Unfortunately the next reference made by this program is to b of zero, which is to will load will also suffer a cache miss and will load to block containing b of zero from memory into the cache. So, that when the time comes to reference a of one next the block which contains a of one is no longer present in the cache it has been replaced by the block containing b of zero through up to b of three. So, the bottom line is that we when we do the analysis for this unfortunate program for the cache under consideration we find out that every single reference is going to be a miss. And some of the misses about we call conflict misses, because they are a result they are not a result of the program starting off with the cache empty, these conflict misses are result of the program making references in such a fashion.

That the references of this program conflict with each other the reference to b of zero replaced block, which would have been to used to satisfy the request for a of one and so on.

(Refer Slide time: 12:00)



Example 2: Cache Hits and Misses

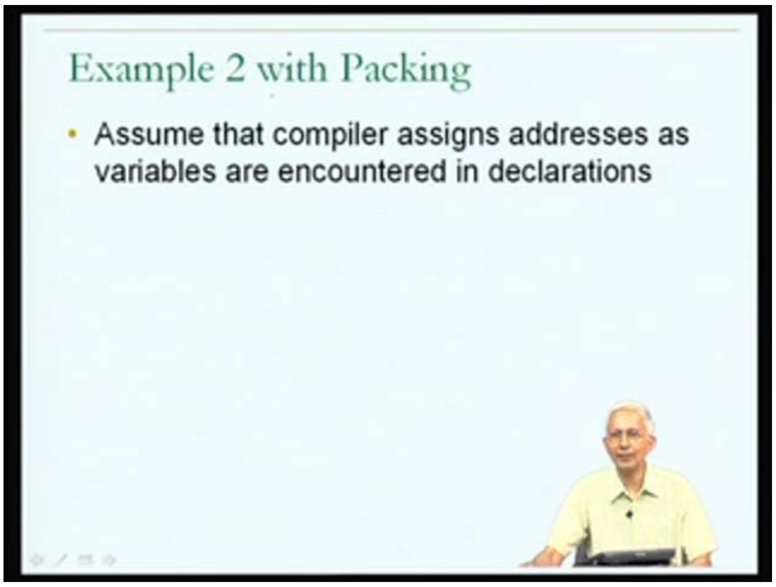
A[0]	0xA000	256	Miss	Cold start
B[0]	0xE000	256	Miss	Cold start
A[1]	0xA008	256	Miss	Conflict
B[1]	0xE008	256	Miss	Conflict
A[2]	0xA010	256	Miss	Conflict
B[2]	0xE010	256	Miss	Conflict
A[3]	0xA018	256	Miss	Conflict
B[3]	0xE018	256	Miss	Conflict
..	..	..	..	
..	..	..	..	
B[1023]	0xFFF8	511	Miss	Conflict

Conflict miss: a miss due to conflicts in cache block requirements from memory accesses of the same program

So, the program is conflicting the references of the program conflict with each other resulting in misses and this is not a problem with, how the program started it is a problem with a structure of the program whether the program is written. The bottom line then is this program will actually; get no benefit from the cache it will end up with a hit ratio of zero percent. In other words every single memory reference every single reference made by this program will result in a memory access under nanoseconds or more and none of the references of this program will benefit from a one cycle one nanosecond cache hit.

And the source of the problem as we saw was that the elements of the arrays a and b, which are accessed in the same order a of zero followed by b of zero followed by a of one followed by b of one. Actually conflict with each other in the sense that they have the same cache index and in a direct map cache they need to occupy the same location in the cache and therefore, one after the other they replace each other resulting into zero percent hit ratio.

(Refer Slide time: 13:00)



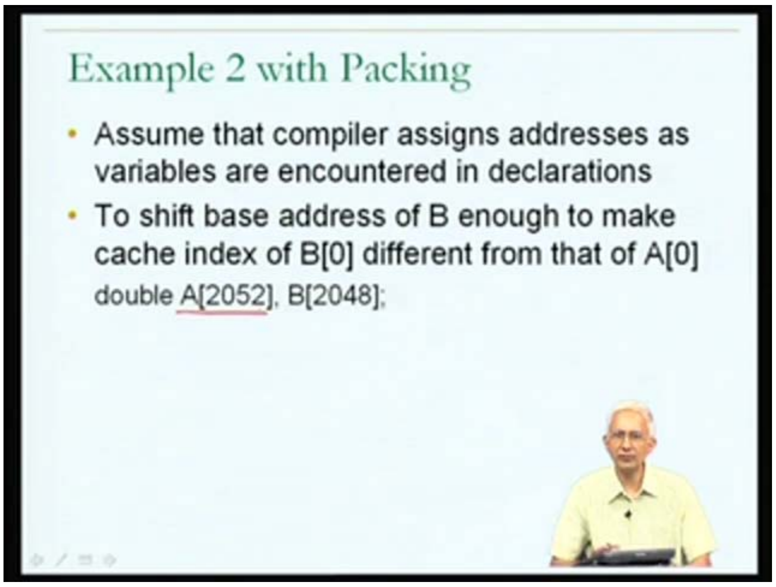
Example 2 with Packing

- Assume that compiler assigns addresses as variables are encountered in declarations

So, the solution which we talked about last time was to actually try to amend this problem correct this problem by causing the base address of let say vector b to be adjusted so, that there is no longer this problem of conflicting in terms of the index value with the base address of vector a. And one way that I suggested this could be done is by assuming or using our knowledge of how compilers lightly to work in that compilers of other once which assign the virtual addresses to the different variables the different arrays of a program. And if we assume that the compilers do so by dealing with a declarations as they are encountered and using consecutive memory addresses for consecutive variables as they as they are declared.

Then we could cause the base address of b to be change for example, by instead of using a declaration bits as double a of 2048, b 2048, we artificially increase the size of the vector a as out lined over here, to 2052.

(Refer Slide time: 14:00)



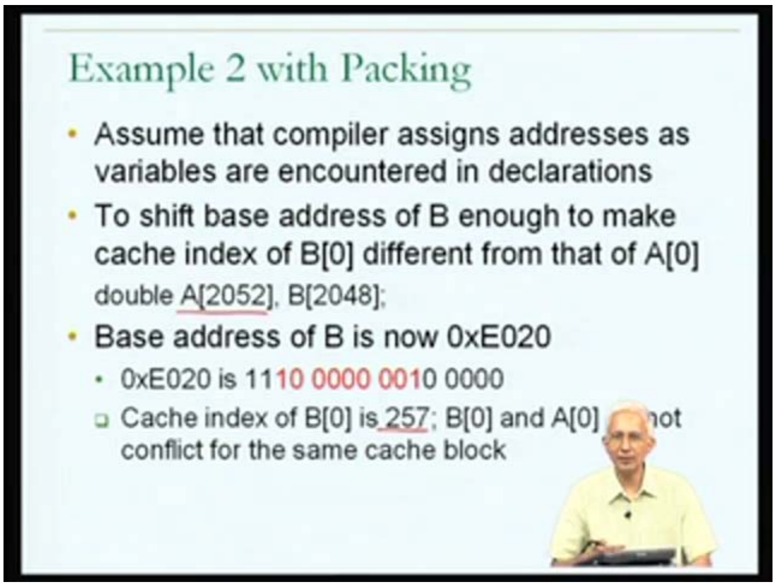
Example 2 with Packing

- Assume that compiler assigns addresses as variables are encountered in declarations
- To shift base address of B enough to make cache index of B[0] different from that of A[0]

```
double A[2052], B[2048];
```

In other words I increase the size of the vector a by four elements, which means I increase the size of the vector a by thirty two bytes which is the size of one cache block. Which we mean that the address of vector b will go up by thirty two and therefore, its index will change by one that will be the net effect, if you go back to the address interpretation by the cache hardware. So, the base address of b would now be e zero two zero rather than e zero zero zero and its index the index value the cache index value for b of zero would now be 257, rather than 256. And therefore, a of zero and b of zero would not conflict for the same cache block and therefore, when a of zero is loaded they will be a cache miss for cold start reasons, after that when b of zero is loaded there will be cache miss for cold start reasons. And b of zero will occupy different location in the cache subsequently, the next reference to loading a of one will get a will get a hit.

(Refer Slide time: 15:00)



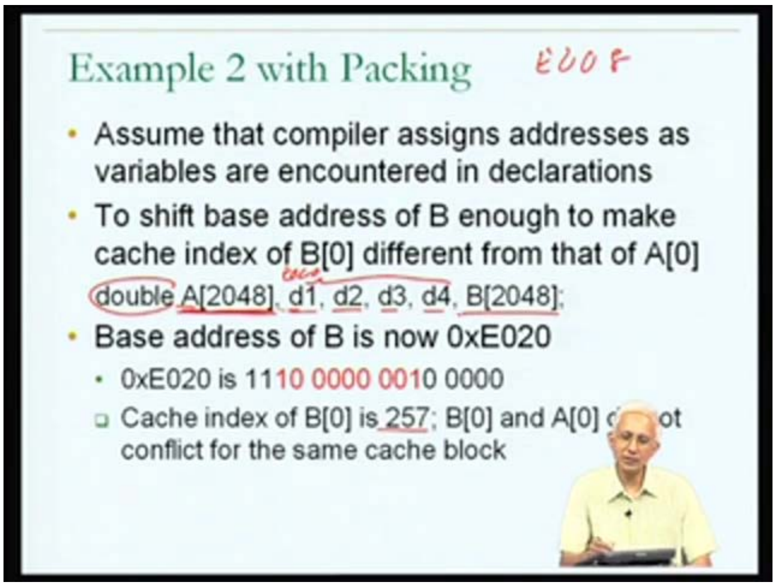
Example 2 with Packing

- Assume that compiler assigns addresses as variables are encountered in declarations
- To shift base address of B enough to make cache index of B[0] different from that of A[0]
  double A[2052], B[2048];
- Base address of B is now 0xE020
  - 0xE020 is 1110 0000 0010 0000
  - Cache index of B[0] is 257; B[0] and A[0] not conflict for the same cache block

Because it is in the same block as a of zero was we get the benefit of spatial locality of reference and the bottom line is there was a hit ratio which is substantially more than the zero percent that we got. And also suggested in alternative way to set this up, rather than artificially increasing the size of the array a if I am assuming that the compiler assigns addresses in order of and encountering the declaration. Then as long as a even if I kept the address of a the size of the declare size of vector a as two thousand and forty eight. If I was to include other variables between, the declaration of a and the declaration of b then I could achieve the same affect of causing the base address of b to b adequately different from the base address of a.

For example, in this variant I have a declared as a vector of size 2048, as it should be b is also declared as vector size to 2048, but in between I declare four variables dummy one dummy two dummy three dummy four it is of type double which means that the total space occupied by these four variables would be thirty two bytes. The base address of d one would be hex e zero zero zero the base address if the compiler assigns consecutive contiguous addresses the base address of d two would be e zero zero eight and so on.

(Refer Slide time: 17:00)



Example 2 with Packing

E208

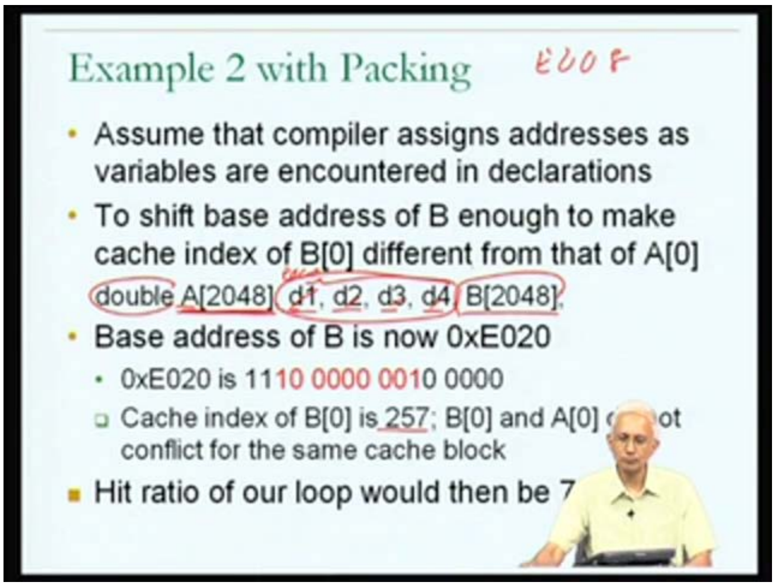
- Assume that compiler assigns addresses as variables are encountered in declarations
- To shift base address of B enough to make cache index of B[0] different from that of A[0]

double A[2048], d1, d2, d3, d4, B[2048];

- Base address of B is now 0xE020
  - 0xE020 is 1110 0000 0010 0000
  - Cache index of B[0] is 257; B[0] and A[0] do not conflict for the same cache block

So, that the base address of b would end up being e zero two zero, which has a sufficiently different index value to not conflict with the base with the vector a. Now, the four variables d one through d four are not variables that my program will use and therefore, even though they may conflict with the vector a. If they had been used the problem does not arise for my for my for this particular program, they variables which are declared and including the program in the merely for the purpose of causing the base address of a of vector b to be such that it does not conflict with the vector a. So, the bottom line in this case would be a substantially, improved hit ratio and you can do the calculation to find out there would be at least seventy five percent great improvement over all zero percent.

(Refer Slide time: 17:00)



Example 2 with Packing

E20F

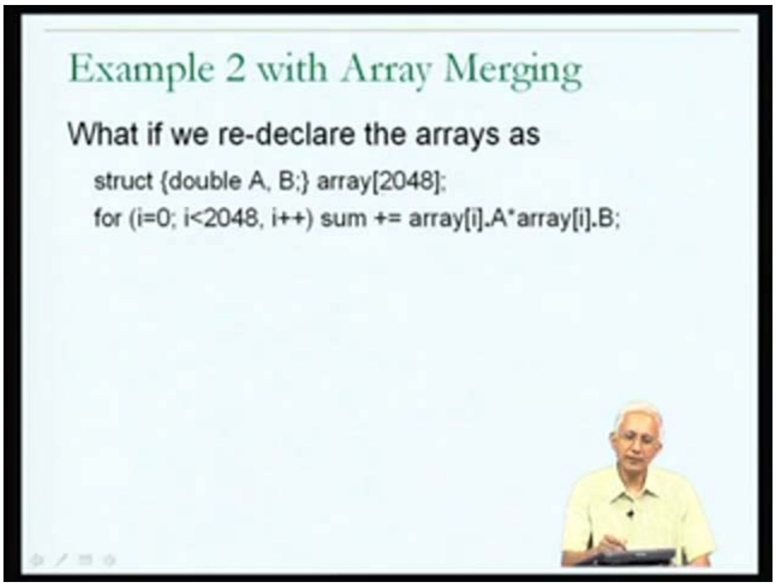
- Assume that compiler assigns addresses as variables are encountered in declarations
- To shift base address of B enough to make cache index of B[0] different from that of A[0]

  double A[2048], d1, d2, d3, d4, B[2048];
- Base address of B is now 0xE020
  - 0xE020 is 1110 0000 0010 0000
  - Cache index of B[0] is 257; B[0] and A[0] do not conflict for the same cache block
- Hit ratio of our loop would then be 7

Now, there are other ways that one could handle this particular problem with this loop and another idea is the idea which is sometimes known as array merging let me just let you know what array merging amongst too. Now, you will recall in the previous version of the program I declared a as a vector of size 2048 and I declare b as a vector of size 2048, but I noted that the way that my program was using the elements of a and the elements of b was that after referring to element a of I it next refer to element b of i. In other words there is this relationship between a of i and b of I, which I could actually cause to be reflected in the declaration of a and b for example, by something like what we have on the screen here.

So, rather than declaring a and b as separate vectors here what I have done is I have declared a and b as being elements of a struct structure and I declare an array of those structure elements.

(Refer Slide time: 18:00)



Example 2 with Array Merging

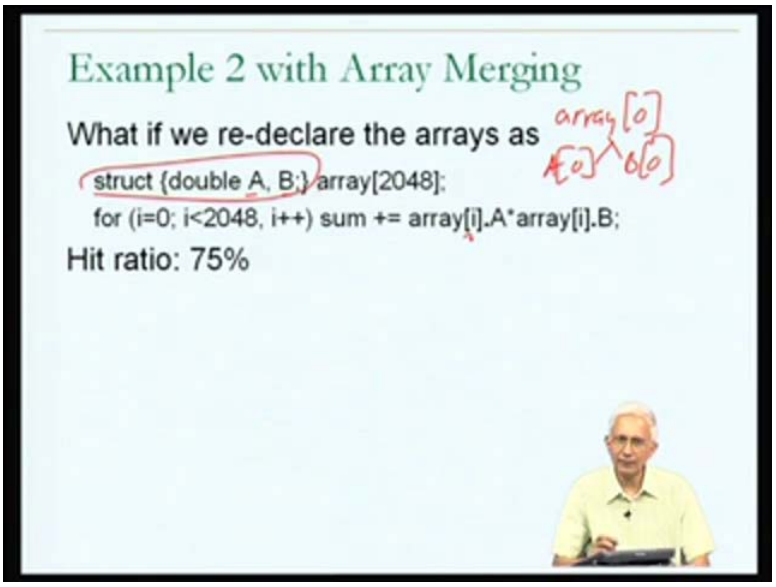
What if we re-declare the arrays as

```
struct {double A, B;} array[2048];
for (i=0; i<2048, i++) sum += array[i].A*array[i].B;
```

Size of that array is 2048, within each element of the struct there is one element of a and one element of b for example, within the element zero of the array I actually have what I use to refer to as a of zero and b of zero and they are package together in element zero of the array by this struct which I have declared. So, now, when I need to right my the vector operation what I have to observe is that I need to multiply the a element of array iteration through this loop I multiply the a element of the a part of the ith element of the array, by the b part of the ith element of the array and then I accumulate them into this some variable. So, it is a minor variant in that the calculation, which is computed is the same as what it was before, but we have change things in that, if you think about the addresses of a and b under this kind of a declaration.

Every element of the array will have an element of a and then element of b and therefore, the element of a and the corresponding element of b will be in consecutive memory locations.

(Refer Slide time: 19:00)


Example 2 with Array Merging

What if we re-declare the arrays as

```
struct {double A, B;} array[2048];
for (i=0; i<2048, i++) sum += array[i].A*array[i].B;
```

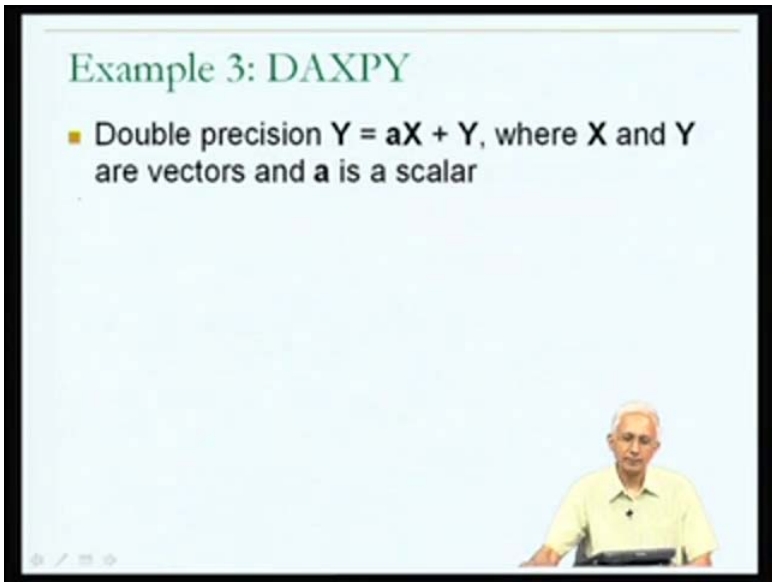
Hit ratio: 75%

And therefore, you doing the kind of calculations we had talked about up to now we would of found out that the element of a zero and the element of b zero would be next to each other in memory. And therefore, they would be in the same cache block and therefore, while access to array I of a might be a cache miss access to array I of b would be a cache hit and similarly, when i go to the next iteration first the elements of a and b would be hits and this is why be come up with an estimate that they would be a 75 percent hit ratio. So, this is an alternative perspective on how one could improve from that zero percent hit ratio for the self same operation of computing the dot product of two vectors.

Now, in some situation it is may not be a good idea to change a declaration like this, because you will as you aware when you write programs often you would want the program to be readable and that the meaning of the program should reflect some. The natural phenomenon or the physics problem or the actual objective for which the program was being written and by artificially packaging elements of a along with elements of b it may become a little bit harder for the person reading or using the program to understand it. But one should bear in mind that the benefits of doing this may could worthwhile to reassess the readability of the program.

And just it remember this is an alternative to the previous technique which had which I had suggested there one carefully at cause the base address of the conflicting vector b to be adjusted, either by adding additional dummy variables in between of a artificially declaring one of the vectors a little bit bigger, than it actually needed to be. Now, moving right along just look at a new example now, the next example I am going to look at a something I will refer to as d a x p y and I will pronouns this as daxpy. Until now the examples, that we used had expanded names which were we could understand the name from the rough understanding of what like vector dot product was here for the first time very clearly this is not going to be the case. But obviously, this is an important operation so important that it has this abbreviated name form.

(Refer Slide time: 21:00)



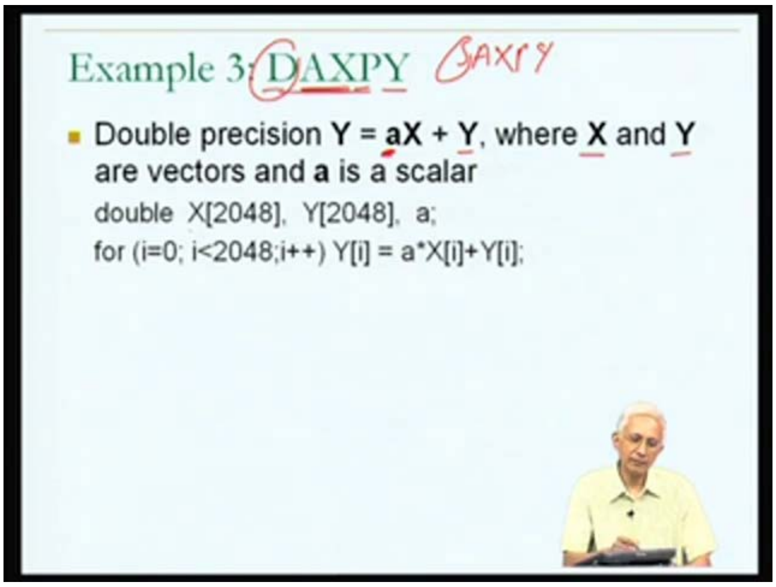
Example 3: DAXPY

- Double precision **Y** = **aX** + **Y**, where **X** and **Y** are vectors and **a** is a scalar

Now, the what the daxpy operation the daxpy program that we are going to talk about does is a double precision operation, where it multiplies a vector x by a scalar value a that is the a x. And then adds to it each element corresponding element of a vector b so, double precision a multiplied by x plus y is what daxpy refers to and one could think of another program which i might called saxpy and as you would imagine this must be referring to a similar operation, but rather than operating on double precision values it operates on single procession or thirty bit floating point values.

So, clearly these must be very important operations and we will refer to daxpy later, but the nature of the operation is quite clear once again we have two vectors a vector x and a vector y. Each time through a loop as we are going to see an element of x is multiplied by some scalar a variable value a right so a is not a vector a is a single valued variable and then the corresponding element of the vector y is added to that product producing the new value of that element of the vector y. So, if I had to view daxpy as a piece of c code as we were doing with vector sum the vector dot product and so on this is what I would see.

(Refer Slide time: 23:00)



Example 3: DAXPY

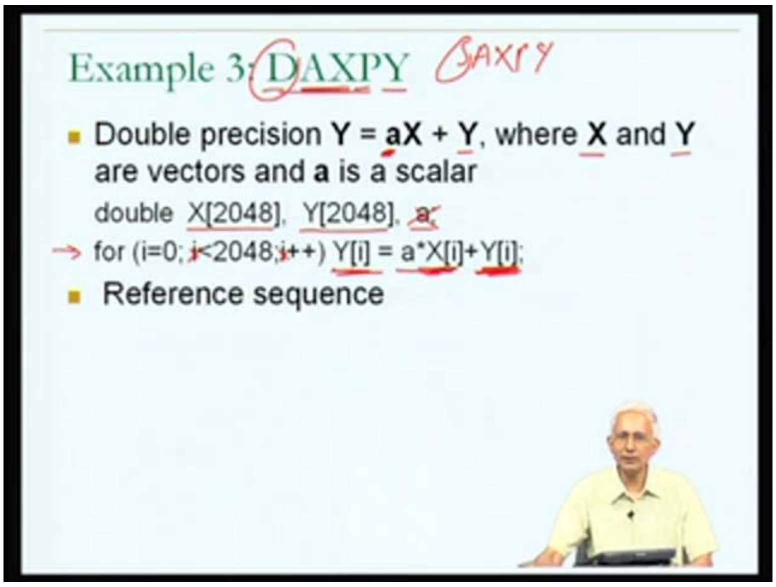
- Double precision **Y** = **aX** + **Y**, where **X** and **Y** are vectors and **a** is a scalar

```
double  X[2048],  Y[2048],  a;
for (i=0; i<2048;i++) Y[i] = a*X[i]+Y[i];
```

Once again I have the vector x which is declare as double and of sum size once again I am using the size 2048, the vector y also declared is a same size and I have the scalar a is not a vector a is a single valued variable. Each times through the for loop one an element the ith element of x is multiplied by a and added to the ith element of y and this becomes a new value of the ith element of y and this is how the vector y is the value of the vector y is computed. So, this operation I would refer to as daxpy so, once again as always we need to start by understanding exactly what the reference sequence is we are going to ignore the variable a. Since a variable a could very easily be stored in a register there is no need is strength through this loop to load the value of a from a memory location into a register the value of a does not change for the duration of this loop.

Similarly, I will assume that the loop index is maintain in register and that is I need concentrate only on the references to x of i y of i X and the additional references time to y of i note that there is I need to load x of I multiplied by a then there is need to load y of i added to this product and then subsequently to store the result into y of i therefore, there are three operations three memory operations in each iteration of the daxpy loop.

(Refer Slide time: 24:21)



Example 3: DAXPY

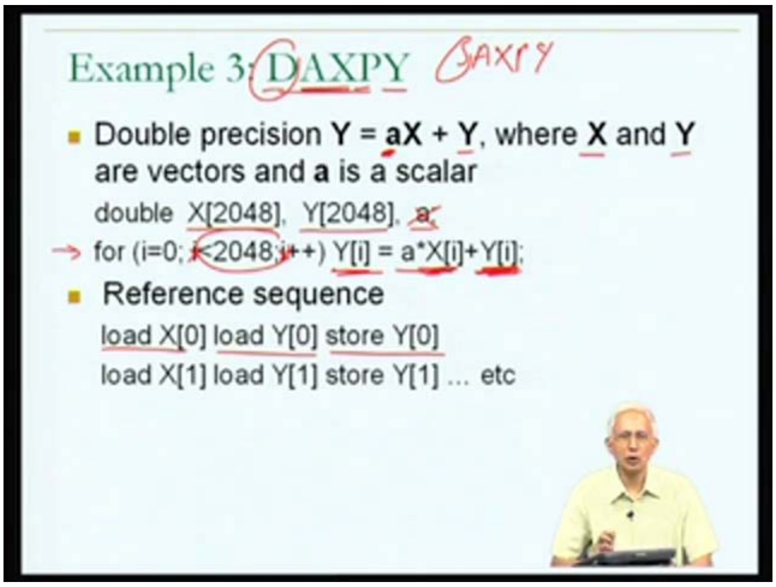
- Double precision **Y** = **aX** + **Y**, where **X** and **Y** are vectors and **a** is a scalar

  double X[2048], Y[2048], a;

  for (i=0; i<2048;i++) Y[i] = a*X[i]+Y[i];
- Reference sequence

So, if I think about the references sequence I look back at the first iteration through the loop. In the first iteration through the loop i is equal to zero therefore, I need to load x of zero, then do the multiplication then I need to load y of zero and then do the addition and then I need to store the result into y of zero. Therefore, these are the three operations in the first iteration of the for loop load x of zero load y of zero and store y of zero.

(Refer Slide time: 25:00)



Example 3: DAXPY

- Double precision **Y** = **aX** + **Y**, where **X** and **Y** are vectors and **a** is a scalar

  double X[2048], Y[2048], a;

  for (i=0; i<2048;i++) Y[i] = a*X[i]+Y[i];
- Reference sequence

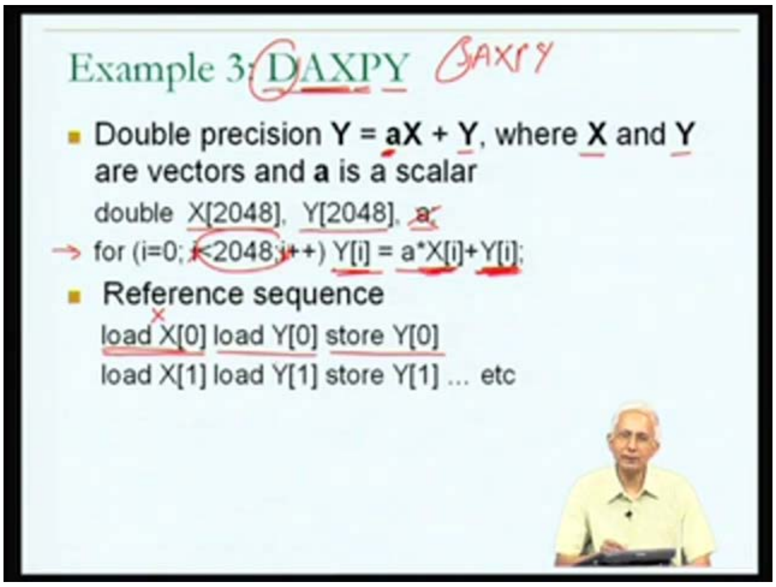
  load X[0] load Y[0] store Y[0]

  load X[1] load Y[1] store Y[1] ... etc

Similarly, in the second iteration when I is equal to one I will load x of one load y of one and store y of one and so on for all of the 2048, iterations. Now, once again we need to make some assumption about the base address of the vector x and the vector y as we now call them. But we could actually quite easily short circuit the operation of going through the full calculation on the bases of drawing a table, but if we at this point we said let us assume that we know enough about how to adjust the base address of x and the base address of y. So, that they do not conflict if I in finding the base address of x and the base address of y I found that they end up having the same index, then I could always adjust the base address of y.

So, let me assume that i have done that adjusting and now I can actually look at this sequence of instructions and try to understand what will happen in terms of hits and misses. So, I might say and again we are going to avoid drawing the whole table since at this point may be possible to do the analysis quite quickly the first reference is to load x of zero and I might say this is guaranteed be a cold start miss.

(Refer Slide time: 26:00)



Example 3: DAXPY

- Double precision **Y** = **a****X** + **Y**, where **X** and **Y** are vectors and **a** is a scalar

  double X[2048], Y[2048], a;

  for (i=0; i<2048; i++) Y[i] = a*X[i]+Y[i];
- Reference sequence

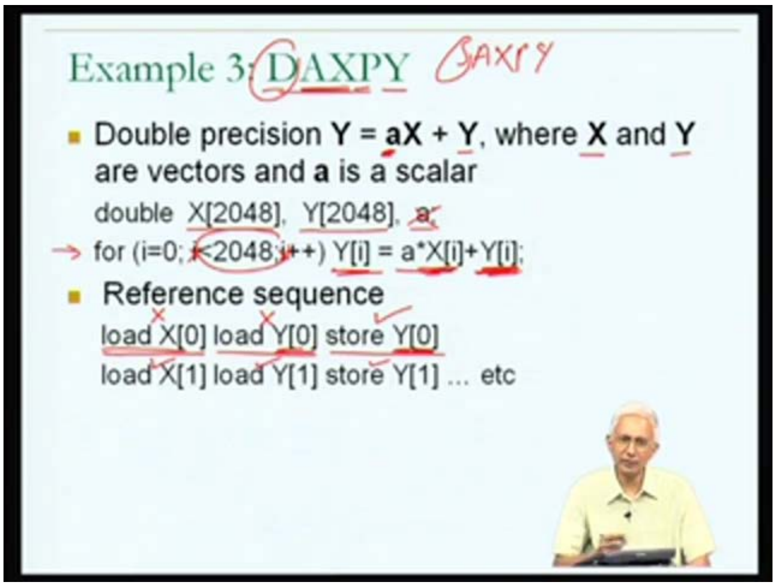
  load X[0] load Y[0] store Y[0]

  load X[1] load Y[1] store Y[1] ... etc

So, no hope for that then there is load y of zero I am assuming it does not conflict with x of zero, but this too is guaranteed to be a cold start miss, then I store y of zero now, I know for I can be quite sure that since a previous reference loaded y of zero y of zero will be in the cache. When the store instruction is executed and therefore, I can be quite confident that the store y of zero will be a cache hit and so this is a new thing which we did not have in our previous examples, a guarantee that they would be a cache hit because, the same variable the same element was accessed in this case in the by the previous instruction.

And move on to the next iteration and I can again see that x of zero x of one x of two and x of three are all going to be in the same cache block and therefore, I would be quite confident that load x of one is going to be a cache hit similarly, load y of one is going to be a cache hit and as always store y of one is going to be a cache hit and I can therefore, proceed with the analysis.

(Refer Slide time: 27:00)



Example 3: DAXPY

- Double precision **Y** = **aX** + **Y**, where **X** and **Y** are vectors and **a** is a scalar

```
double X[2048], Y[2048], a;
for (i=0; i<2048;i++) Y[i] = a*X[i]+Y[i];
```

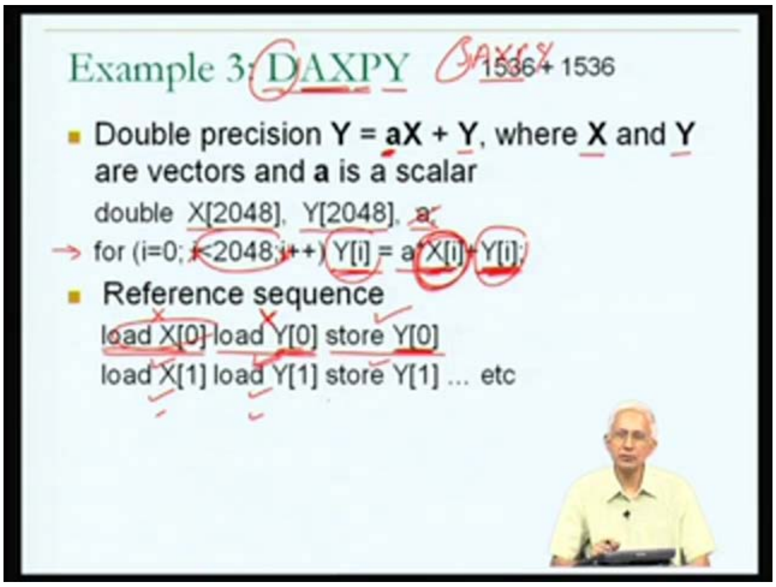
- Reference sequence

load X[0] load Y[0] store Y[0]
load X[1] load Y[1] store Y[1] ... etc

But instead of drawing the table as I said let us try to do the analysis in terms of calculating how many hits I am going to get for the loads of x of i how many hits I am going to get for the loads of y of i and how many hits i am going to get for the stores of y of i. I know that each I know that I am going to have to load x of i 2048, times I know that the first one would be a miss but the next three would be hits subsequently the attempt to load x of four would be a miss. So, I now that one out of every four references to x of i is going to be a miss, but the remaining three out of four, because of spatial locality of reference will be hits. Therefore, as far as the 2048, loads of the vector elements of x of i are concerned I can say that they would be 1536 six hits.

What I am going to do is calculate the hit ratio in the upper right hand corner I can then move to my assessment of what about the loading of the 2048 elements of y of i and once again I know in the iteration they will be a cold start miss in second iteration, because of spatial locality they will be a hit similarly, in the third and forth iterations they will be hits. And just as in the case of the loads of x of i for the loads of y of i they would be 1536 hits out of 2048 references.

(Refer Slide time: 28:00)



Example 3: DAXPY

SAXPY 1536 + 1536

- Double precision **Y** = **aX** + **Y**, where **X** and **Y** are vectors and **a** is a scalar

```
double X[2048], Y[2048], a;
for (i=0; i<2048;i++) Y[i] = a*X[i]+Y[i];
```

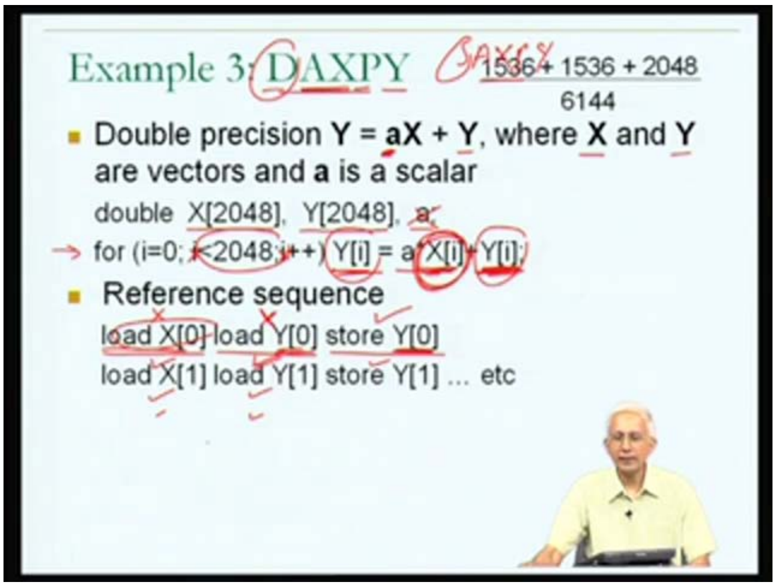
- Reference sequence

load X[0] load Y[0] store Y[0]

load X[1] load Y[1] store Y[1] ... etc

So, the number of hits that I expect are 1536 for the references to a and other 1536 for the references to y what about the last sequence of references that I making the stores to the elements of y I know that all of those 2048, stores are going to be hits. So, I just add it 2048 for the guaranteed hits as far as the store instructions are concerned how do I calculate the hit ratio I divide the total number of hits by the total number of memory references what is the total number of memory references that is going to be given by the number of iterations, which is 2048, multiplied by the number of memory references per iteration which is three. Therefore, the denominator of my hit ratio calculation will be 2048, multiplied by three which is six thousand one hundred and forty four.

(Refer Slide time: 29:00)



Example 3: DAXPY

SAXPY 1536 + 1536 + 2048

6144

- Double precision **Y** = **aX** + **Y**, where **X** and **Y** are vectors and **a** is a scalar

```
double X[2048], Y[2048], a;
for (i=0;i<2048;i++) Y[i] = a*X[i]+Y[i];
```

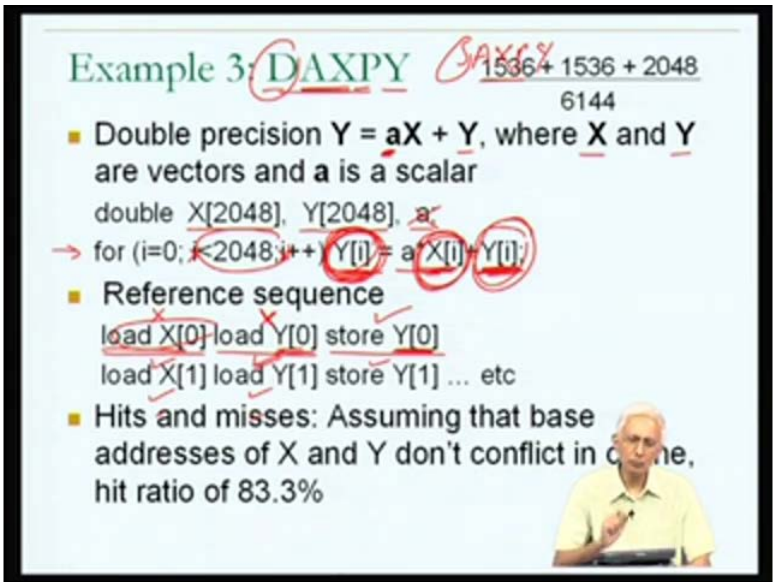
- Reference sequence

  load X[0] load Y[0] store Y[0]

  load X[1] load Y[1] store Y[1] ... etc

So, I actually know that the hit ratio will be this value and if you do the calculation you will find out that under the assumption that the base addresses of x and y do not conflict in the cache you come up with this quick assessment that the hit ratio will be 83.3 percent. And this was higher than the 75 percent that we got in the previous example, because of this slide difference in the nature of the program where we had one third of all references being guaranteed hits by the very nature of the I am sorry I should be circling the stores the stores are all going to be hits, because of the fact that the element which is being stored was just accessed by the previous instruction which was a load instruction.

(Refer Slide time: 30:00)


Example 3: DAXPY

1536 + 1536 + 2048
6144

- Double precision Y = aX + Y, where X and Y are vectors and a is a scalar

  double X[2048], Y[2048], a;

  for (i=0; i<2048;i++) Y[i] = a*X[i]+Y[i];
- Reference sequence

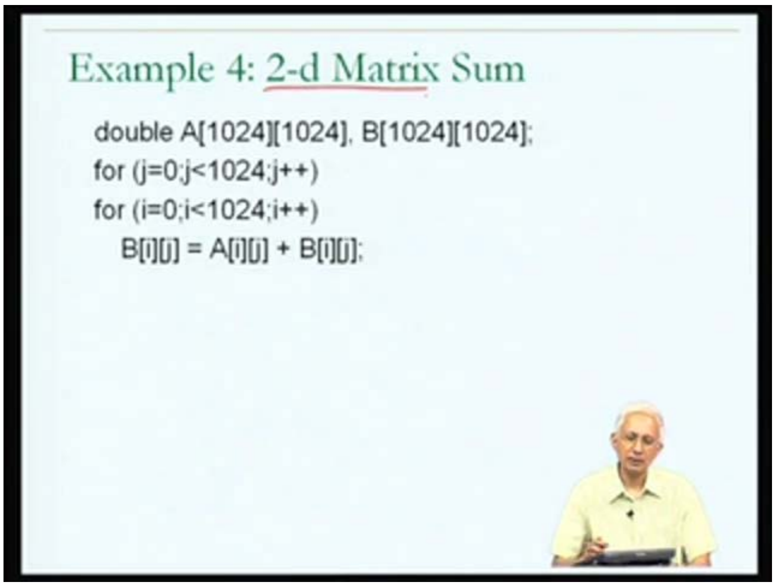
  load X[0] load Y[0] store Y[0]

  load X[1] load Y[1] store Y[1] ... etc
- Hits and misses: Assuming that base addresses of X and Y don't conflict in cache, hit ratio of 83.3%

So, daxpy was quite easy for us to analyze and those no need for us to draw the table we would like to move in this direction what to we what you are observing is that we are moving away from our requirement of actually knowing the base addresses or actually doing the calculation of the index value for each of the elements with this experience of having work through two or three simple examples. And with this kind of quick analysis, we get a very good idea what to expect when the program will actually execute on a given piece of cache hardware.

So, we can actually move on to another example, quite rapidly I did want to just remind you that we will be encountering daxpy again that is I want to introduce the word the term daxpy we will be encountering again in the not too distant future. Now, let us move on to a slightly more interesting example this example is an example, in which we are not dealing with ordinary one dimensional vectors any more, but we are actually dealing with a two dimensional matrix and the operation is quite simple we have two dimensional matrices and we want to add them element by element.

(Refer time: 31:00)



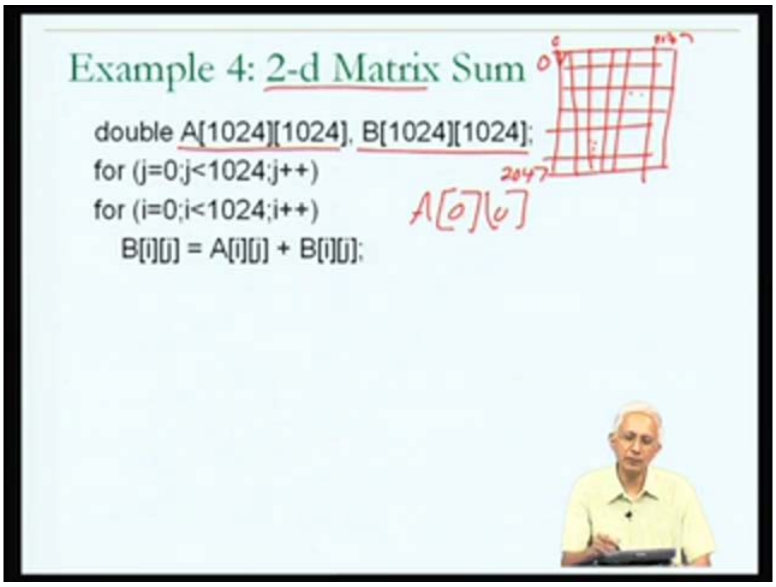
Example 4: 2-d Matrix Sum

```
double A[1024][1024], B[1024][1024];
for (j=0;j<1024;j++)
for (i=0;i<1024;i++)
   B[i][j] = A[i][j] + B[i][j];
```

So, that the structure of the program is as shown over here I have two double precision two dimensional matrices. So, each matrix has 1024 rows each of which is made up of 1024 columns the picture that you can have in mind when thinking of either the matrix a or the matrix b is, rather than when we thought out of a vector we thought of a linear entity when we think of a matrix a two dimensional matrix we think of an element which has many rows. So, the lines which I have drawn are meant to represent the different rows so the 2048, rows which would be numbered from row zero up to row 2047 and each row is comprised of 2048 columns in the columns are numbers from zero up to 2047.

So, this is the picture that one could have in mind when thinking of let say the matrix a or the matrix b and any particular element of the matrix is refer to using two entasis the row index and the column index. For example, to refer to that element over there if this is the matrix a then I refer to it as a of zero which means in row zero in other words specifically in column zero.

(Refer Slide time: 32:00)

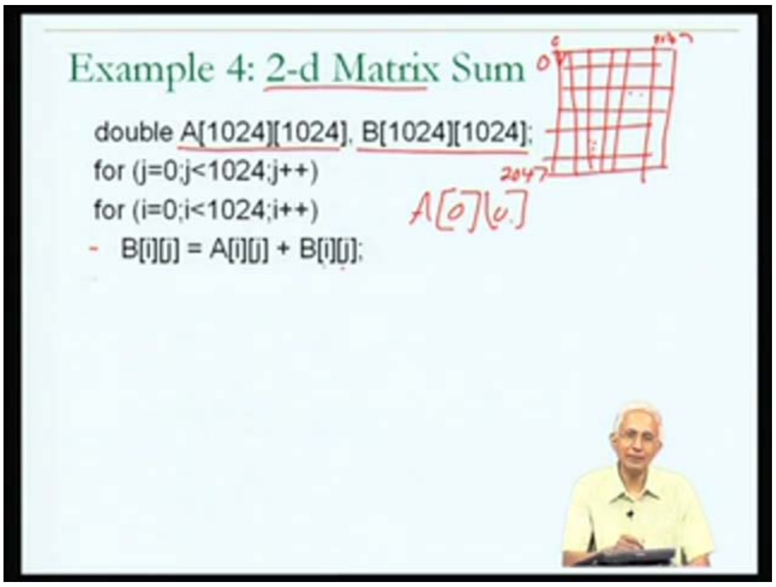


Example 4: 2-d Matrix Sum

```
double A[1024][1024], B[1024][1024];
for (j=0;j<1024;j++)
for (i=0;i<1024;i++)
   B[i][j] = A[i][j] + B[i][j];
```

So, the element of row zero column zero of the matrix a and that is notation which use in c exactly the same so, I have two such matrices and i want to add them element by element and this particular example I am putting the result of doing this addition as a new value of the matrix b. So, in this particular case we need to have a w nested for loop and each time through the w nested for loop an element of a the ith the element of a in the ith row in the jth column is added to the corresponding element of b. In other words the element of b from the ith row and jth column the addition is done and this becomes a new value of that particular element of b. In other words the ith row and jth column of b takes on this sum so you will notice that the number of times that this statement within the loop is going to executed is 1024 multiplied by 1024.

(Refer Slide time: 33:00)

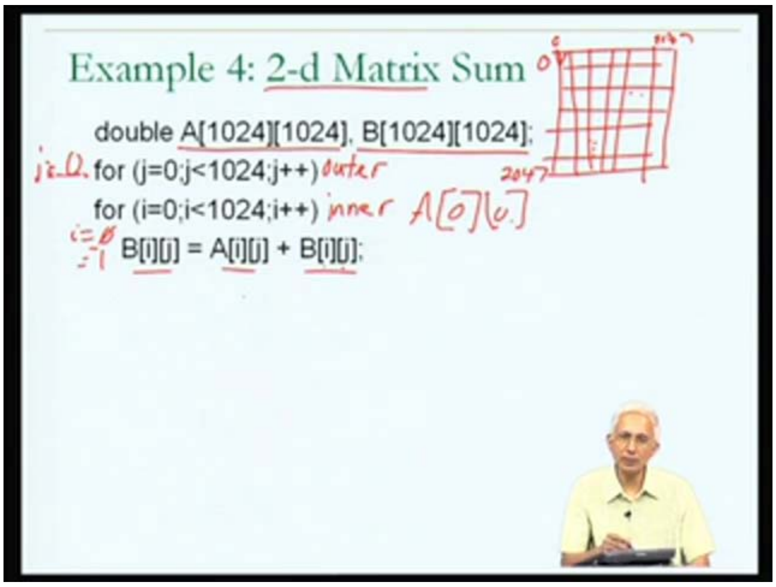

Example 4: 2-d Matrix Sum

```
double A[1024][1024], B[1024][1024];
for (j=0;j<1024;j++)
for (i=0;i<1024;i++)
  B[i][j] = A[i][j] + B[i][j];
```

Since number of elements in this to the each of these two dimensional matrices is 1024 multiplied by 1024. So, as before we can run through this example, by trying to layout the order in which the different references occur. So, I try to derive the reference sequence so and looking at this just bear in mind that we have a situation where there are w nested for loops in other words, if consider the for loop on the outside I will refer to the j loop as the for loop on the outside. So, initially the for loop the j loop will cause j to have a value of zero, then we enter the inner loop which is the i loop in this case so, terminology which I have used this I talk about the j loop as the outer loop and the i loop as the inner loop within the inner loop we start by setting i equal to zero.

And therefore, the first memory operation that happens is loading a of 0 0 after that we load b of 0 0 and then we store we do the addition and we store the result in b of 0 0, then we step back and increment i by one so, we go from i equal to zero to i equal to one j is still zero. And therefore, the next vector element of the next matrix element that we deal with is adding a of one zero to b of one zero.

(Refer Slide time: 34:00).

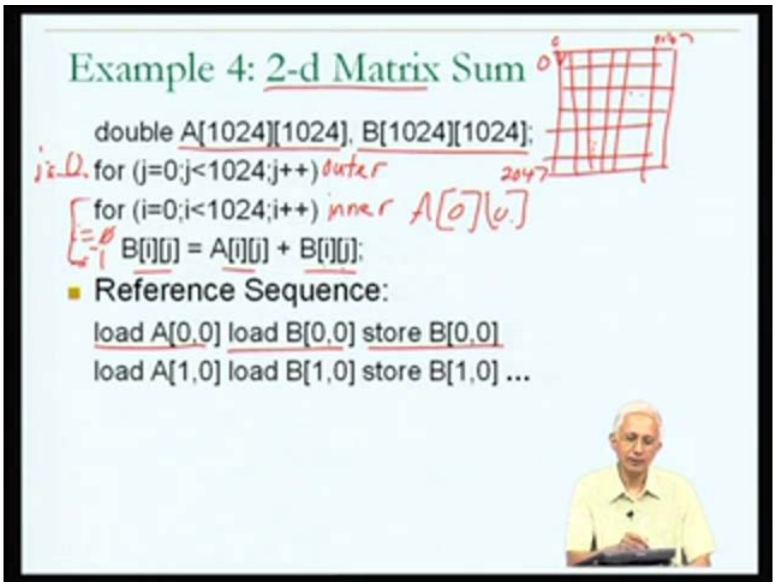


Example 4: 2-d Matrix Sum

```
double A[1024][1024], B[1024][1024];
for (j=0;j<1024;j++)
    for (i=0;i<1024;i++)
        B[i][j] = A[i][j] + B[i][j];
```

So, we have an operation of loading a of one zero loading b of one zero doing the addition and storing the sum into p of one zero. So, if I was to and so on so as I write out the complete reference sequence I have the first iteration through this w nested loop which loaded a of 0 0 0 loaded b of 0 0 did the addition and stored the result in b of 0 0 after that remember that the value of i change to one.

(Refer Slide time: 35:00)



Example 4: 2-d Matrix Sum

```
double A[1024][1024], B[1024][1024];
for (j=0;j<1024;j++)
    for (i=0;i<1024;i++)
        B[i][j] = A[i][j] + B[i][j];
```

- Reference Sequence:

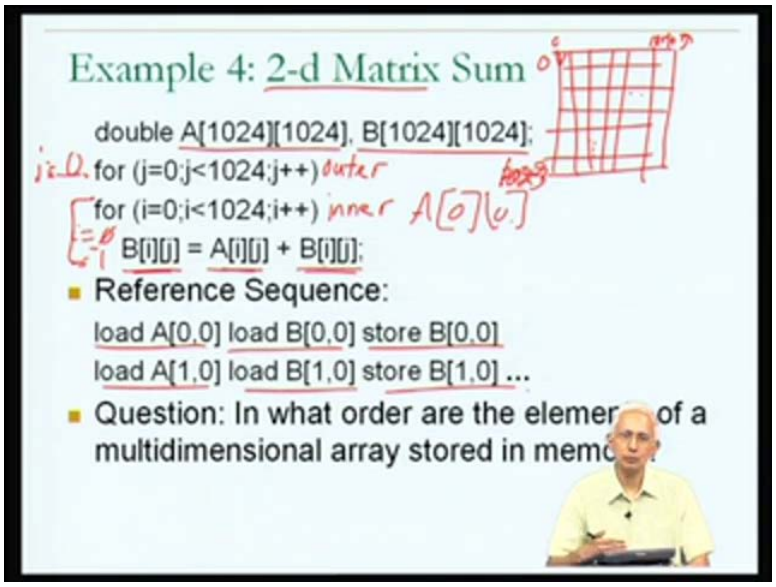
  load A[0,0] load B[0,0] store B[0,0]

  load A[1,0] load B[1,0] store B[1,0] ...

Because we were the inner loop will be iterated 2024, times I am sorry this is thousand and twenty four and not two thousand and forty seven so these number should not have been thousand and forty seven but thousand and twenty three. So, in the second iteration through the w nested loop I load a of one zero remember that I has been incremented to one so, it will be a of one zero and then I load b of one zero do the addition and put result in b of one zero and so on.

So, there will be 1024 multiplied by 1024 iterations through this and there will be corresponding number of references in the reference sequence. Now, to actually proceed from here things are little bit more complicated then what we talked about in the case of the vector examples that we looked at because when we talked about the vector examples it was very clear. That the assumptions we were making was that the neighbors of a of for example, a neighbor of a of zero would be a of one and the neighbors of a of one would be a of zero and a of two.

(Refer Slide time: 36:00)



Example 4: 2-d Matrix Sum

```
double A[1024][1024], B[1024][1024];
for (j=0;j<1024;j++)
  for (i=0;i<1024;i++)
    B[i][j] = A[i][j] + B[i][j];
```

- Reference Sequence:

  load A[0,0] load B[0,0] store B[0,0]

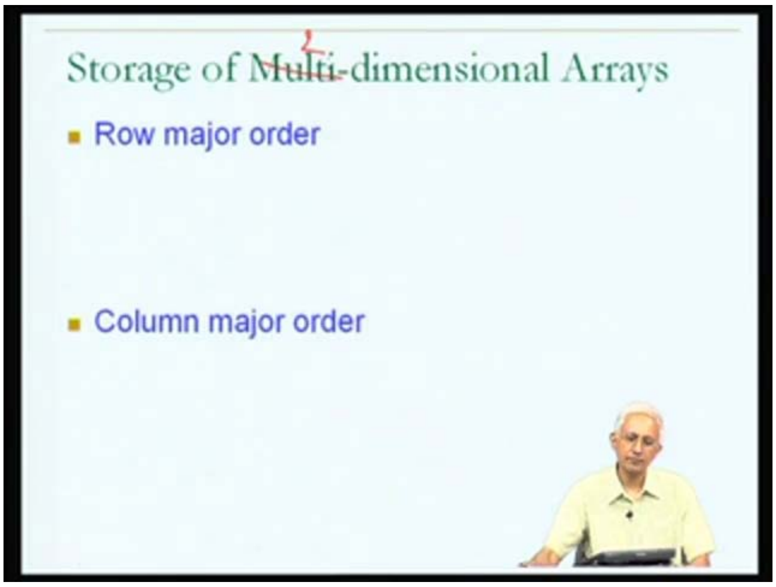
  load A[1,0] load B[1,0] store B[1,0] ...

- Question: In what order are the elemen… of a multidimensional array stored in memo…

We assume that the vector elements will lay out consequentially in memory, but when we have a two dimensional structures such as what is shown over here we do have to ask the question on the screen, in what order all the elements of a multidimensional array. Such as these two dimensional array which we are currently looking at stored in memory and you will quickly realize, that there are at least I am sorry there are two possibilities unlike the case of the vector where it was clear that there was only one option consecutive vector elements in consecutive memory locations.

Here there are two possibilities you could either have the neighbor of a of 0 0 being a of zero one and the neighbor of a of zero one could be a of zero two alternatively it could compiler could lay out the array elements in memory matrix elements in memory. So, the neighbor of a of 0 0 is a of one zero and the neighbor of a of one zero is a of two zero and so on. So, there at least these two possibilities which is why we will spend the whole slide just giving some terminology to these possibilities.

(Refer Slide time: 37:00)

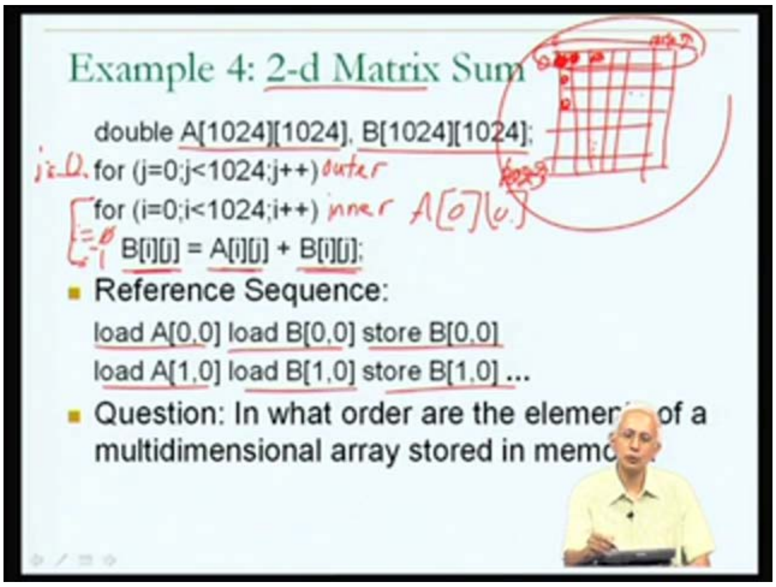


Storage of ~~Multi~~ 2-dimensional Arrays

- Row major order
- Column major order

So, this is a the topic which is sometimes known as the storage order of multidimensional arrays in our particular example we are looking at two dimensional arrays, but something similar what happen for three four or higher dimensional arrays. The two options which I just talked about are in fact known as the row major order and the column major order and the row major order was the first, which we had looked at the idea that. The elements of the first row of the matrix are stored in consecutive memory locations and they are then followed by the elements of the second row of the matrix etcetera, as I had suggested to start off with. Remember the first row of the matrix a is the elements circled and therefore, the neighbor of a of 0 0 is a of zero one and the neighbor of a of zero one is 0 0 two and so on.

(Refer Slide time: 38:00)



Example 4: 2-d Matrix Sum

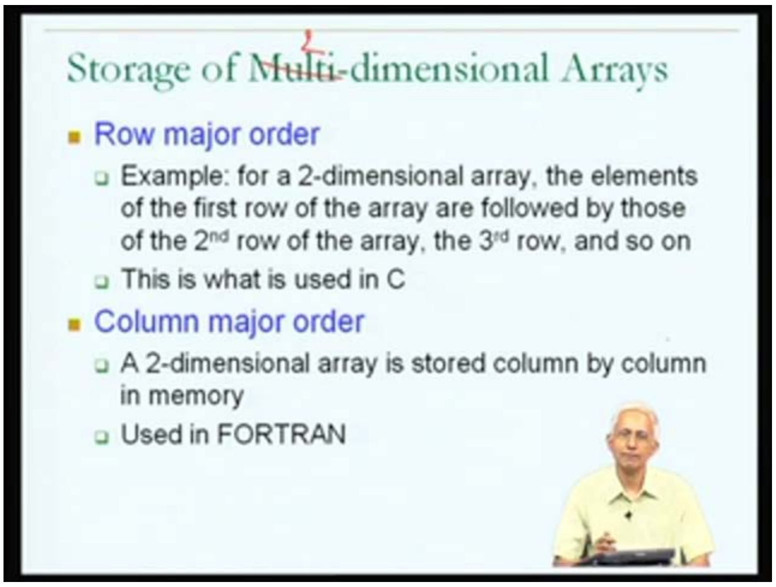
```
double A[1024][1024], B[1024][1024];
for (j=0;j<1024;j++)
  for (i=0;i<1024;i++)
    B[i][j] = A[i][j] + B[i][j];
```

- Reference Sequence:
  load A[0,0] load B[0,0] store B[0,0]
  load A[1,0] load B[1,0] store B[1,0] ...
- Question: In what order are the elemen of a multidimensional array stored in memo

Which means that in consecutive locations in memory one would find the different elements of the first row of the matrix and this would be followed by the elements of the second row of the matrix and so on. In some sense one could talk about the storage order as being row by row, if one looked at how they laid out in memory first the first row, then the second row then the third row remember that the memory that we have the main memory has a linear ordering of the bytes from zero up to two power given the number of bytes in memory.

So, that is why the idea of row major order is to view the ordering of the array matrix elements when stored in memory as being row by row hence the name row major and this is in fact the convention used by c the c language convention used by c compilers. The alternatives what is known as column major order the one which have spoken of second, we will actually store the two dimensional array column by column in memory in other words first the first column zero of the matrix a followed in memory by the second column of matrix a and so on.

(Refer Slide time: 39:00).

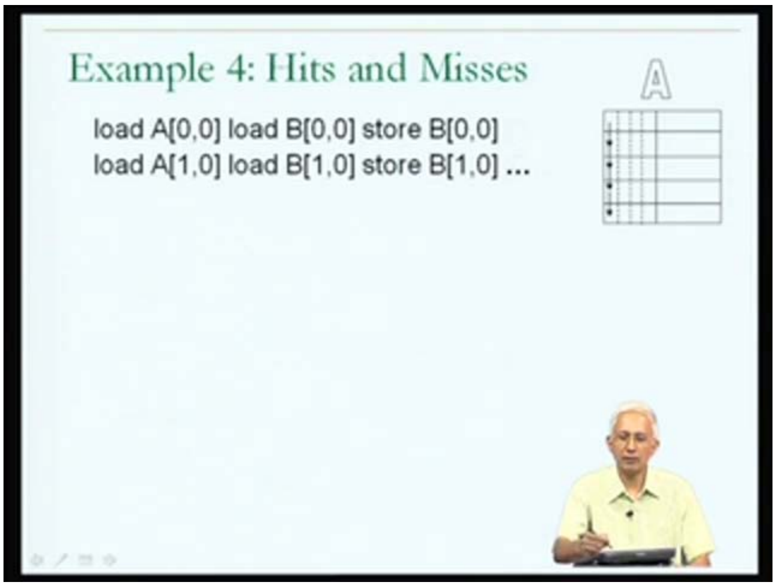


Storage of ~~Multi~~-dimensional Arrays

- Row major order
  - Example: for a 2-dimensional array, the elements of the first row of the array are followed by those of the 2nd row of the array, the 3rd row, and so on
  - This is what is used in C
- Column major order
  - A 2-dimensional array is stored column by column in memory
  - Used in FORTRAN

And this order two is not ignored this order used by some programming languages such as Fortran so this is the interesting thing to note, which of these two storage orders is used for your multidimensional arrays of your program will depend on what language you have used to write the program. Now, coming back to our example of the matrix two dimensional matrix sum, we have understood the reference order. The reference order was load a of 0,0 load b of 0,0and then store b of 0,0 followed by load a of 1,0 load b of 1,0 store b of 1,0 and so on. In other words the order in which the elements of a here I have a diagram showing you the matrix a the first access a of 0,0 and next we access a of 1,0 if you looking at the order in which the elements of matrix a are being accessed by my program.

(Refer Slide time: 40:00)



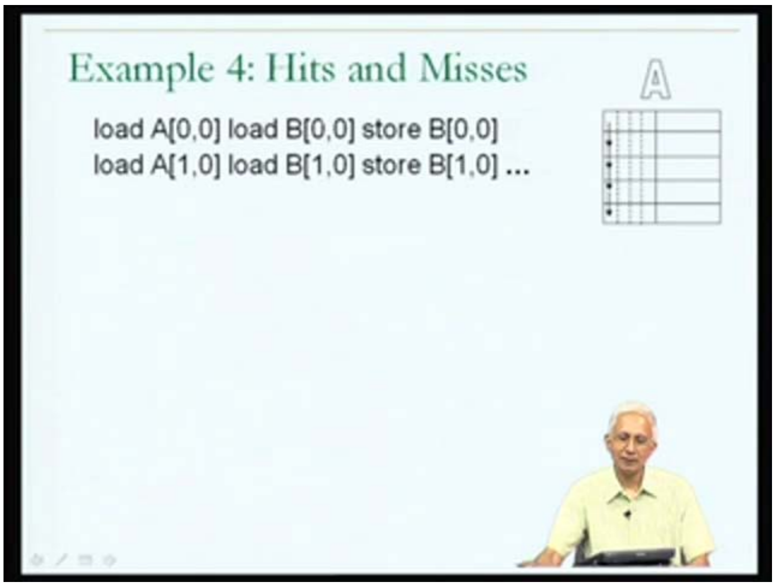
Example 4: Hits and Misses

load A[0,0] load B[0,0] store B[0,0]
load A[1,0] load B[1,0] store B[1,0] ...

So, the elements of matrix have been or reference by my program column by column that is the observation which I would make and similarly, I can observe that the elements of matrix b first I access b of 0,0 then b of 0,0 once again. But the next element of the matrix b which I access is b of 1,0 which means that in the case of b two in the case of b also, we are accessing the elements of d column by column which I we does which I will designate by this first the first column after that the second column and so on. Now, you would have notice that the program that we were we started with seems to have been programmed in c and we just learned in the previous slide, that as far as c is concerned the row major ordering of array elements is used.

In other words if you looked at consecutive memory locations you would find first the first row of matrix a and then after that the second row of matrix a and so on.

(Refer Slide time: 41:00)



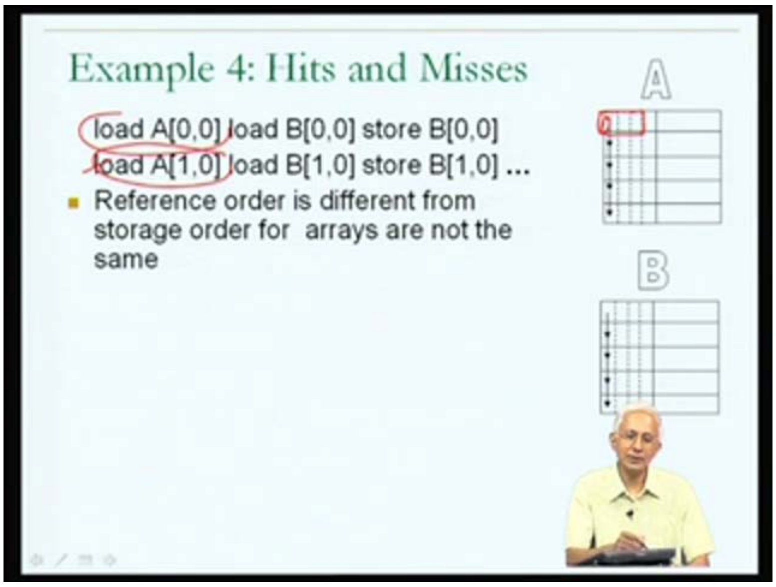
Example 4: Hits and Misses

load A[0,0] load B[0,0] store B[0,0]
load A[1,0] load B[1,0] store B[1,0] ...

What is that mean it means that we are the program is accessing the array elements column by column, but the array elements are stored in memory row by row what is that mean in terms of the cache performance. What this tells us is that first of all a of 0,0 it was referenced and that is going to be a cold start miss at some sometimes soon after that a of 1,0 is reference, but we know that the block cache block containing a of 0,0 given that this is a program written in c will contain the neighbors of a in the row containing a. And therefore, the cache block will contain a of 0, 0 a of 0, 1 a of zero two and a of zero three whereas, the second reference I have made in this program is to load a of 1, 0 and therefore, this will not be a cache hit due to spatial locality of reference, but rather it will be another cache miss.

So, therefore, this is a negative we notice that this is the program in the way that we have written it is not going to be benefit from spatial locality of reference it is basically going to have a of 0,0 a cold start miss a of 1,0 a cold start miss.

(Refer Slide time: 42:00).



Example 4: Hits and Misses

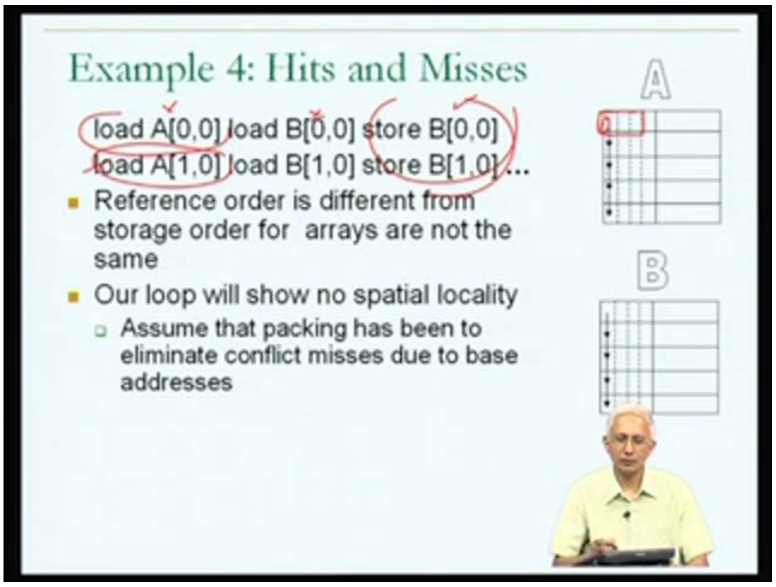
load A[0,0] load B[0,0] store B[0,0]
load A[1,0] load B[1,0] store B[1,0] ...

- Reference order is different from storage order for arrays are not the same

Similarly, for b b of 0, 0 is going to be a cold start miss store b of 0, 0 will be a hit, because b of 0, 0 was referenced by the previous instruction. That other then the store instructions all the remaining instructions will not benefit from spatial locality of reference and this is the negative this means that our program is not was not written with knowledge of cache in mind. So, bottom line is our loop will show no spatial locality of reference a loop is showing temporal locality of reference notice that all the references to the store instructions benefit from locality.

But they benefit benefitting from temporal locality not from spatial locality and practically our program will not really as far as we can see show any spatial locality of reference. Even if we assume that a and b are not conflicting with each other.

(Refer Slide time: 43:00)



Example 4: Hits and Misses

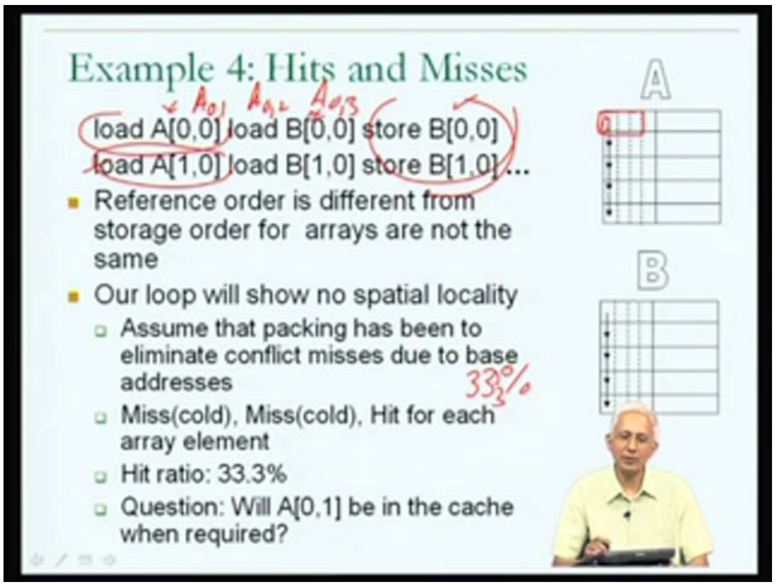
load A[0,0] load B[0,0] store B[0,0]
load A[1,0] load B[1,0] store B[1,0] ...

- Reference order is different from storage order for arrays are not the same
- Our loop will show no spatial locality
  - Assume that packing has been to eliminate conflict misses due to base addresses

So, in trying to come up with hit ration calculation, we can make the assumptions that we can use packing or some other technique to make sure that the accesses to a and b do not conflict with each other due to base address type of a problem. But the situation is going to be that in the iterations we will have a miss for a of 0, 0 a miss of b of 0, 0 and then the hit for the store, but that is going to happen for each array element. And therefore, we suspect that we may end up with the hit ratio of about one third or 33 percent which is much lower than the kinds of hit ratio we were getting with the previous examples that you looked at remember seventy five percent eighty three percent and so on.

So, this is not that is not seem to be a well written program from the perspective of comparison with other programs that we were able to analyze. There was of course, one issue which I would want to raise the suggestion is that we have hit ratio of 33 percent or 33.3 percent to be to be more accurate.

(Refer Slide time: 44:00)



Example 4: Hits and Misses

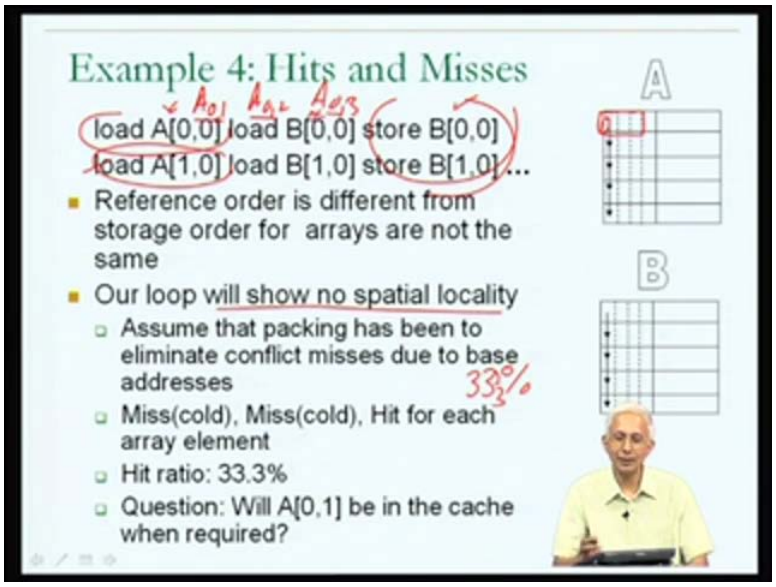
load A[0,0] load B[0,0] store B[0,0]
load A[1,0] load B[1,0] store B[1,0] ...

- Reference order is different from storage order for arrays are not the same
- Our loop will show no spatial locality
  - Assume that packing has been to eliminate conflict misses due to base addresses
  - Miss(cold), Miss(cold), Hit for each array element
  - Hit ratio: 33.3%
  - Question: Will A[0,1] be in the cache when required?

But there is an underline question which may be would be worth thinking about we notice that when a of 0, 0 was missed the block containing a of 0, 0 would be loaded into the cache and the block containing a of 0, 0 would have contained a of 0, 0 along with a of 0, 1 a of 0, 2 and a of 0, 3 which are the three neighbors in the row as far as the two dimension matrix a is concerned. So, the block will contained those four array elements unfortunately, the next few iterations of my loop do not refer to a of 0, 1 a of 0, 2 of a 0, 3 which is why we said that the program does not show good spatial spatial locality of reference.

But we do know that some time later this program is going to access a of 0, 1 a of 0, 2 and a of 0, 3 hence, this question much later in time when this program executes the loop where it loads a of zero one will a of zero one still be in the cache can we calculate whether a of 0, 1 will still be in the cache if that is the case, then a of zero one the load of a of 0, 1 will actually be a hit the load of b of 0, 1 by the same token would be a hit and so on.

(Refer Slide time: 45:00)



Example 4: Hits and Misses

load A[0,0] load B[0,0] store B[0,0]
load A[1,0] load B[1,0] store B[1,0] ...

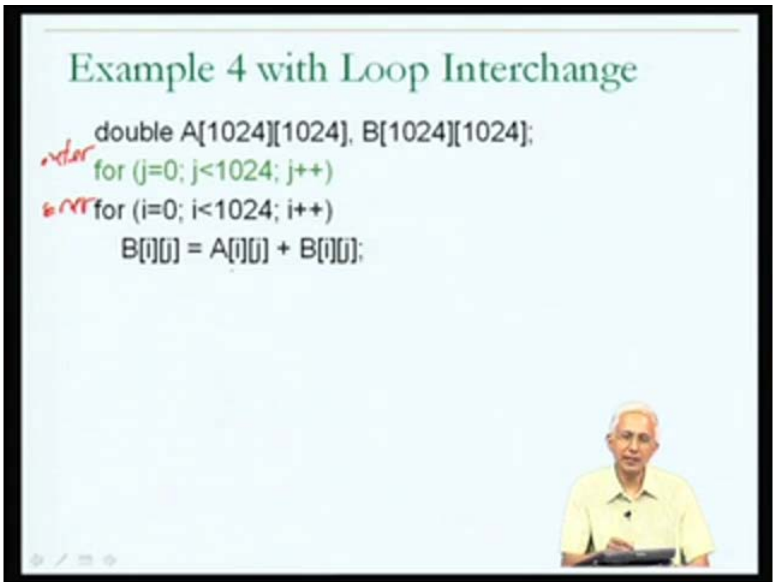
- Reference order is different from storage order for arrays are not the same
- Our loop will show no spatial locality
  - Assume that packing has been to eliminate conflict misses due to base addresses
  - Miss(cold), Miss(cold), Hit for each array element
  - Hit ratio: 33.3%
  - Question: Will A[0,1] be in the cache when required?

And therefore, the hit ratio may be substantially better than 33.3 percent and we might actually benefit from spatial locality of reference. So, the question is how do we analyze whether later on in time when our program does in fact try to load a of 0, 1 the block containing a of 0, 1 which was fetched by the first load instruction in the program will still be in the cache. In one way to reason about this rather quickly is just to bear in mind that in our particular program, we have a situation where there are row, if you look at the size of each column we have a situation where the size of the each column is 1024 elements and therefore, you should note that when we will come to loading a of 0, 1after we have finished going through the inner loop once.

And therefore, the questions has to be analyzed in terms of how much of the cache would have been filled by the time we come back to the outer loop and change the value of j to one. And with this calculation we will be able to get an answer to the question about whether a of 0, 1 would be present in the cache or not and based on that calculation we have to figure out whether the hit ratio might in fact be higher than that we have calculated over here I will leave that for you to analyze on the side. So, with this we have an understanding that, if multidimensional arrays they may be something a little bit more complicated and there we may have to go back to the source code the or c program in order to prove it in order to make the hit ratio higher.

Now, the simple idea which we are go try for to do for to analyze this something which is called loop interchange and the idea of loop interchanges do you remember that we started off with a vector two dimensional matrix sum the program look like this. And you will remember that the outer loop was the j loop and the inner loop this is what we had earlier was the i loop and it seems to the loop we were accessing a of i j and b of i j.

(Refer Slide time: 47:00)

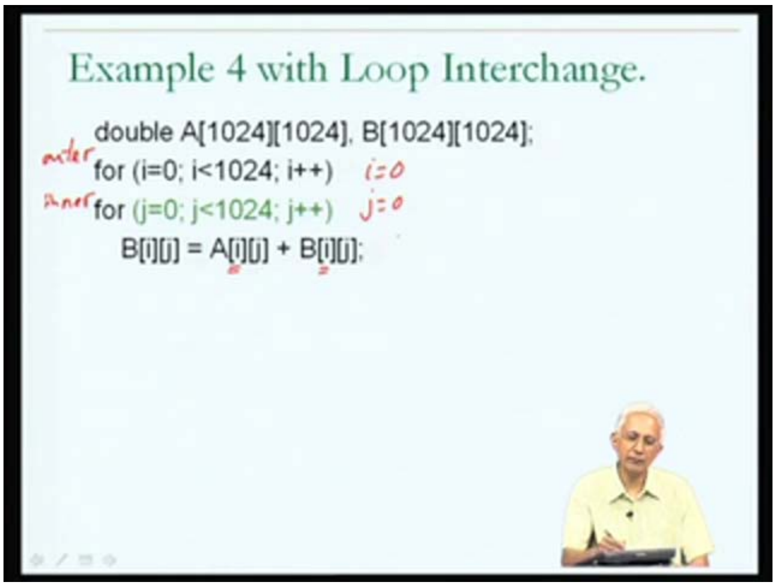


Example 4 with Loop Interchange

```
double A[1024][1024], B[1024][1024];
for (j=0; j<1024; j++)
for (i=0; i<1024; i++)
    B[i][j] = A[i][j] + B[i][j];
```

Now, what if we realize that the problem with this loop was that we were going through the through the two dimensional matrix is column by column and that we could correct for that by just going through the two dimensional matrices row by row. There is nothing the definition of matrix sum two dimensional matrix sum, which says that we have to go through the matrices column by column. We could just as well have gone through them row by row and one way to achieve that is actually to interchange the j loop with the i loop. In other words instead of having the j loop as the outer loop and the i loop is the inner loop I could modify the program so that the i loop is the outer loop and the j loop is the inner loop which is what i have done.

(Refer Slide time: 48:00)



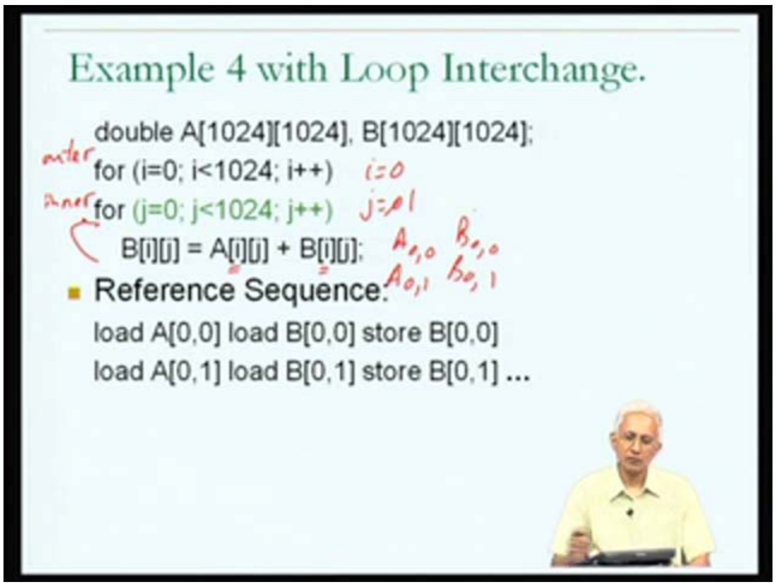
Example 4 with Loop Interchange.

```
double A[1024][1024], B[1024][1024];
outer for (i=0; i<1024; i++)    i=0
inner for (j=0; j<1024; j++)    j=0
        B[i][j] = A[i][j] + B[i][j];
```

Now, notice that the outer loop is now a loop in which be I vary the value i I being the first index into the matrix a and the matrix b and the inner loop is now the j loop which means, that initially the first time you go through the outer loop i will be equal to zero, then you go to the inner loop and j will be equal to zero and therefore, you operate on a of 0, 0 and b of 0, 0. After that you go to the second iteration of the inner loop where the value of j is change to one and you operate on a of 0, 1e b of 0, 1 and notice that this is now stepping through the two matrices row by row.

Therefore, bearing in mind that this was a c program and that in a c program the two dimensional matrices would be stored row by row by just correctly interchanging the two the two loops the inner loop the j loop and the i loop along the lines of what we see here. We would be causing the program to have a reference sequence which is the same as the storage order for the array elements.

(Refer Slide time: 49:00)



Example 4 with Loop Interchange.

```
double A[1024][1024], B[1024][1024];
for (i=0; i<1024; i++)
for (j=0; j<1024; j++)
    B[i][j] = A[i][j] + B[i][j];
```

- Reference Sequence:

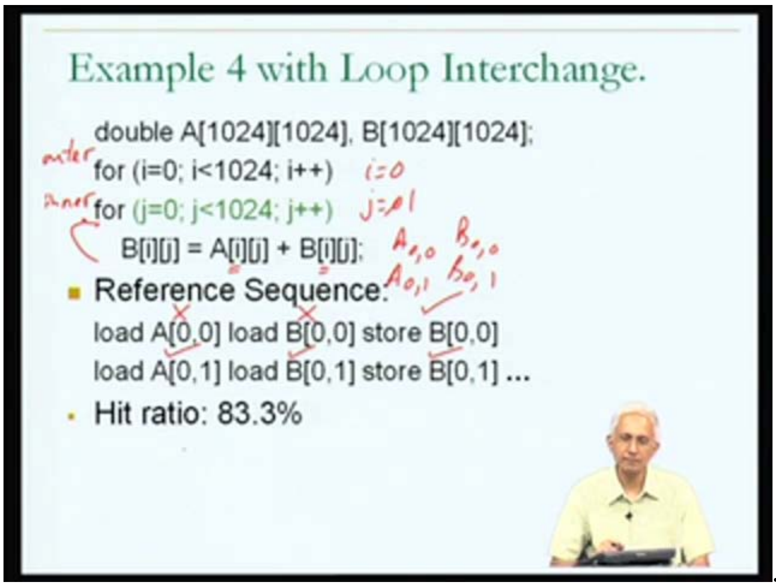
  load A[0,0] load B[0,0] store B[0,0]

  load A[0,1] load B[0,1] store B[0,1] ...

outer
inner
i=0
j=0 1
$A_{0,0}$ $B_{0,0}$
$A_{0,1}$ $B_{0,1}$

So, we now have a situation where the modified program and all that I have done is to interchange the line four i line by the four j line that is only change that was made. The reference sequence will now be load a of 0, 0 load b of 0, 0 store b of 0, 0 followed by load a of 0, 1 load b of 0, 1 store b of 0, 1. And now, we notice that the program will benefit from spatial locality of reference the load b a of 0, 0 will be a cold start miss load b of 0, 0 will be a cold start miss store b of 0, 0 will be a hit load a of 0, 1 will be a hit, because of spatial locality of reference. Similarly, load b of 0, 1 and similarly, load a of 0, 2 load a of 0, 3 will be hits and therefore, we get a substantially improved hit ratio in fact the 83.3 percent which we had the best kind of hit ratio that we have seen from our previous examples.

(Refer Slide time: 50:00)



Example 4 with Loop Interchange.

```
double A[1024][1024], B[1024][1024];
for (i=0; i<1024; i++)
for (j=0; j<1024; j++)
    B[i][j] = A[i][j] + B[i][j];
```

- Reference Sequence:

  load A[0,0] load B[0,0] store B[0,0]

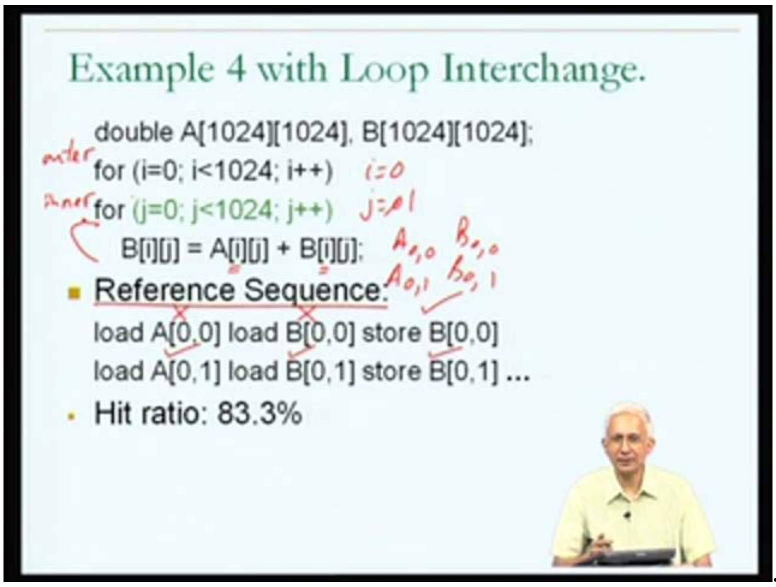
  load A[0,1] load B[0,1] store B[0,1] ...

- Hit ratio: 83.3%

So, clearly in time to deal with programs that have multidimensional arrays or the two dimensional arrays that we are looking at right now it is important to have the order in which the array elements are accessed carefully thought out. Otherwise you will a have situation whether reference sequence differs from the storage order of the array elements and therefore, spatial locality will not become available and the program will not benefit from the spatial locality properties benefits of the underlying cache hardware.

Now, this idea of loop interchange is interesting the idea that one could interchange the i loop and the j loop, but this is similar to something that we have seen earlier. You will recall that when we are talking about instructions scheduling I interchanged instructions at the machine language level. And at that point I had point made special mentioned that it may not always be safe to just arbitrarily interchange to instructions since the meaning of the program could change.

(Refer Slide time: 51:00)



Example 4 with Loop Interchange.

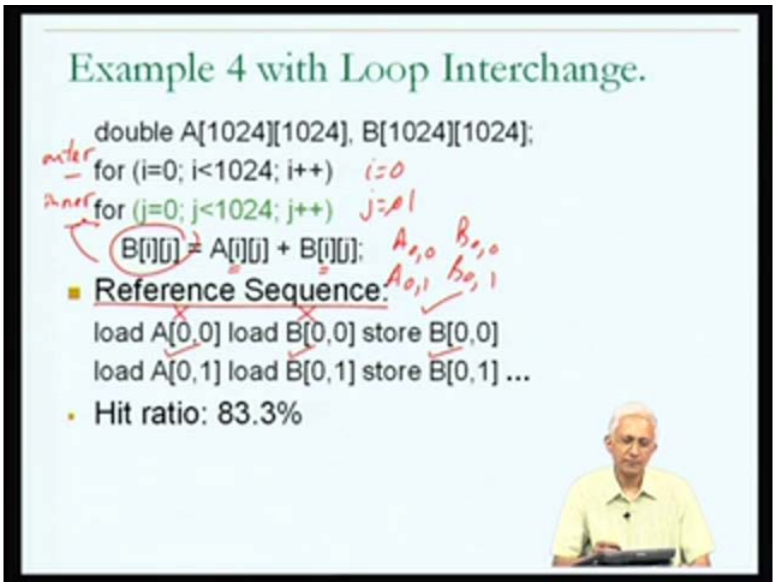
```
double A[1024][1024], B[1024][1024];
for (i=0; i<1024; i++)
for (j=0; j<1024; j++)
    B[i][j] = A[i][j] + B[i][j];
```

- Reference Sequence:
  load A[0,0] load B[0,0] store B[0,0]
  load A[0,1] load B[0,1] store B[0,1] ...
- Hit ratio: 83.3%

The net effect of executing the modify program could be different from the net effect of executing the preexisting program and therefore, we do have to ask the questions of whether one can just interchange the i loop and the j loop without changing the meaning of the program. Now, for this particular example the two dimensional matrix sum you would quite easily be able to satisfy yourself, that there is no harm from interchanging the i loop and the four. And the j loop in that each time through this loop by just dealing with array element a of i j of b and the corresponding element of a adding them together and modifying the array matrix b and that therefore, whether I go through the matrices row by row or column by column the net effect is going to be same on the result.

In other words the final value of the matrix b and that the only difference is that the version in which I have interchange the loops will run faster, because of the cache improved cache behavior. But they could well be loops in which it is not safe to do this kind of an interchange and maybe we have to understand loop interchange little bit more carefully in order to figure out.

(Refer Slide time: 52:00)



Example 4 with Loop Interchange.

```
double A[1024][1024], B[1024][1024];
for (i=0; i<1024; i++)
for (j=0; j<1024; j++)
    B[i][j] = A[i][j] + B[i][j];
```

- Reference Sequence:

  load A[0,0] load B[0,0] store B[0,0]

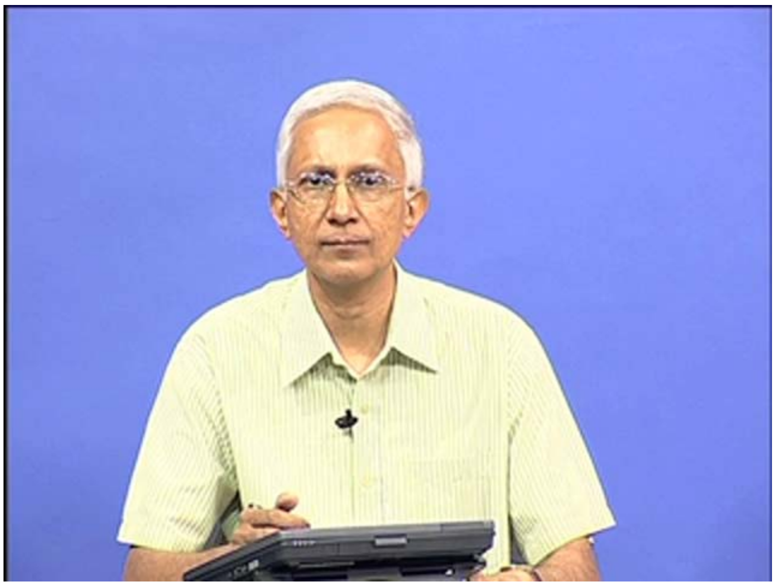
  load A[0,1] load B[0,1] store B[0,1] ...

- Hit ratio: 83.3%

Whether loop interchange is a generally applicable phenomenon or whether we have to do it much more carefully with little bit more caution and I would ask you to just check satisfy yourself for this particular loop it is perfectly safe to do the loop interchange is absolutely no danger of the program doing something different. Now, in the next lecture we will actually start by moving to an example where it is not safe to interchange the loop.

So, we will try to analyze why it is not safe before proceeding to some other examples in this sequence, in this class you will recall that we have looked at some useful additions to our reporta of techniques to improve cache performance. We had seen the idea of packing or changing the base address of a matrix of a vector in the previous lecture, we added to that the idea of actually merging two vectors into a single vector to get the benefit of spatial locality across the two array references.

(Refer Slide time: 53:00)



And we were looking at multidimensional matrices and realize that it may be important to make sure, that the storage order in terms of how the compiler is assigning addresses to the elements of the multidimensional array should be well aligned with the order in which the program references the array elements, otherwise once again the program may not benefit from spatial locality of reference and we stop here for today. Thank you.